A theory of bisimulation for a fragment of concurrent ML with local names

Alan Jeffrey
CTI, DePaul University
243 South Wabash Ave
Chicago IL 60604, USA
ajeffrey@cs.depaul.edu

Julian Rathke
COGS, University of Sussex
Brighton BN1 9QH, UK
julianr@cogs.susx.ac.uk

Abstract: Concurrent ML is an extension of Standard ML with π-calculus-like primitives for multi-threaded programming. CML has a reduction semantics, but to date there has been no labelled transitions semantics provided for the entire language. In this paper, we present a labelled transition semantics for a fragment of CML called $\mu\nu$CML which includes features not covered before: dynamically generated local channels and thread identifiers. We show that weak bisimulation for $\mu\nu$CML is a congruence, and coincides with barbed bisimulation congruence. We also provide a variant of Sangiorgi's normal bisimulation for $\mu\nu$CML, and show that this too coincides with bisimulation.

Keywords: concurrency, higher-order functions, bisimulation, local names

1 Introduction

Reppy's [23] Concurrent ML is an extension of Standard ML [25] with concurrency primitives based on Milner's CCS [16] and π-calculus [17, 18].

Reppy [23] and Berry, Milner and Turner [2] provided CML with a reduction semantics, based on the SML language definition [25]. This semantics allows proofs of many important properties such as subject reduction, but does not directly support a notion of program equivalence, which is important for rewrites such as compiler optimizations.

Ferreira, Hennessy and Jeffrey [6] provided a labelled transition system semantics for a fragment of CML, and showed that the resulting theory of bisimulation was a congruence, and could therefore be used to justify equational rewriting. The fragment of CML covered, however, was more restrictive than Reppy's reduction semantics, and in particular did not treat two important features of the language: channel generation, and thread identifiers. In this paper, we show how these features can be modelled using labelled transition semantics.

Channel generation is an important primitive of CML: it allows new communication channels to be created dynamically, and for their scope to be controlled in the style of the π-calculus. Much of the power of CML rests on channel generation, for example it is used in Reppy's coding of recursion into λ_{cv}. Many languages, such as Fournet *et al.*'s join calculus, [7, 9] Boudol's blue calculus [4], Thomsen's CHOCS [30] and Sangiorgi's higher-order π-calculus [26] include encodings of the λ-calculus using unique names. This paper provides the first direct characterization of program equivalence for the λ-calculus together with π-style concurrency.

The full language of CML contains thread identifiers. In particular, the type for the spawn primitive which creates new threads is given by

$$\mathsf{spawn} : (\mathsf{unit} \to \sigma) \to \sigma\,\mathsf{thread}$$

That is, spawn takes an inactive thread and sets it running concurrently with the new thread. The result of this command is the identifier name of the new thread, which can then be used to block waiting for another thread to terminate, using:

$$\mathsf{join} : \sigma\,\mathsf{thread} \to \sigma$$

Calling $\mathsf{join}\,t$ causes the current thread to wait for t to terminate with some value v, which is then returned.

Ferreira, Hennessy and Jeffrey's treatment of CML ignored thread identifiers entirely. Their type for thread spawning was simply:

$$\mathsf{spawn} : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$$

Thread identifiers have received less attention in the literature than channels, largely because they can be easily expressed in terms of channels. In this paper, we provide a semantics for thread identifiers, partially because they do exist in concurrent ML, but also because they help simplify the lts semantics. This use of thread identifiers is similar to the use of function definitions in Fournet *et al.*'s join calculus, and object pointers in Gordon and Hankin's [11] concurrent object calculus, but with the important difference that thread identifiers can contain actively executing code rather than just functions or objects. They can also be seen as a restricted form of

$$\frac{}{\Gamma;\Delta,n:\sigma\vdash n:\sigma}\qquad\frac{\Gamma;\Delta\vdash e:\sigma_1\qquad\Gamma,x:\sigma_1;\Delta\vdash t:\sigma_2}{\Gamma;\Delta\vdash \mathsf{let}\,x=e\,\mathsf{in}\,t:\sigma_2}$$

$$\frac{\Gamma;\Delta\vdash v_1:\sigma\,\mathsf{thread}\qquad\Gamma;\Delta\vdash v_2:\sigma\,\mathsf{thread}}{\Gamma;\Delta\vdash v_1=v_2:\mathsf{bool}}\qquad\frac{\Gamma;\Delta\vdash v_1:\sigma\,\mathsf{chan}\qquad\Gamma;\Delta\vdash v_2:\sigma\,\mathsf{chan}}{\Gamma;\Delta\vdash v_1=v_2:\mathsf{bool}}$$

$$\frac{\Gamma;\Delta\vdash v_1:B\qquad\Gamma;\Delta\vdash v_2:B}{\Gamma;\Delta\vdash v_1=v_2:\mathsf{bool}}\qquad\frac{\Gamma;\Delta\vdash v:\mathsf{bool}\qquad\Gamma;\Delta\vdash t_1:\sigma\qquad\Gamma;\Delta\vdash t_2:\sigma}{\Gamma;\Delta\vdash \mathsf{if}\,v\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2:\sigma}$$

$$\frac{\Gamma;\Delta\vdash v:\mathsf{unit}}{\Gamma;\Delta\vdash \mathsf{chan}\,v:\sigma\,\mathsf{chan}}\qquad\frac{\Gamma;\Delta\vdash v:\sigma\,\mathsf{chan}*\sigma}{\Gamma;\Delta\vdash \mathsf{send}\,v:\mathsf{unit}}\qquad\frac{\Gamma;\Delta\vdash v:\sigma\,\mathsf{chan}}{\Gamma;\Delta\vdash \mathsf{recv}\,v:\sigma}$$

$$\frac{\Gamma;\Delta\vdash v:\sigma\,\mathsf{thread}}{\Gamma;\Delta\vdash \mathsf{join}\,v:\sigma}\qquad\frac{\Gamma;\Delta\vdash v:\mathsf{unit}\to\sigma}{\Gamma;\Delta\vdash \mathsf{spawn}\,v:\sigma\,\mathsf{thread}}$$

Figure 1: Thread type inference rules (not showing the usual simply typed λ-calculus rules)

$$\frac{;\Delta,n:\sigma\,\mathsf{thread}\vdash t:\sigma}{\Delta,n:\sigma\,\mathsf{thread}\vdash n[t]}\qquad\frac{}{\Delta\vdash\mathbf{0}}\qquad\frac{\Delta\vdash C_1\qquad\Delta\vdash C_2}{\Delta\vdash C_1\,\|\,C_2}\qquad\frac{\Delta,n:\sigma\vdash C}{\Delta\vdash \nu n:\sigma\,.\,C}$$

Figure 2: Configuration type inference rules

Cardelli and Gordon's [5] ambients, where the ambient tree is flat and names are used linearly.

The authors made initial steps towards the current labelled transition semantics for local names in [15]. We proposed there a novel transition system which incorporated a notion of privacy. We adopt the same technique for modelling local names in the current setting; however, the proof techniques for establishing congruence of bisimulation differ greatly. In the previous paper we utilized a variant of a proof technique known as Howe's method [14, 10]. Unfortunately, this is not available to us here and, instead we adapt Sangiorgi's [26] trigger semantics from the higher-order π-calculus. The approach we develop here generalizes the trigger encodings and their corresponding correctness proofs to accommodate the functional setting. This is achieved by a introducing a hierarchy of operational semantics based on type-order and establishing correctness throughout the hierarchy. We also make use of a novel 'bisimulation up to' technique [27] based on confluent reduction to simplify and structure the correctness proofs.

The remainder of the paper is organized as follows: in the next section we present the fragment of CML which contains the features of interest to us and define a type system, reduction semantics and notion of observational equivalence for the language. In section 3 we describe our labelled transition system semantics and offer justification for our transition rules by demonstrating a contextuality result. Bisimulation equivalence for our language is also presented here. Section 4 is given over to establishing that bisimulation is a congruence. We follow this up with a much simpler labelled transition semantics for which bisimulation equivalence coincides with the equivalence of Section 3. Finally we conclude with some closing remarks about related and future work.

2 A core fragment of CML

We examine the language $\mu\nu$CML, which is a subset of Reppy's [23] concurrent functional language CML, given by extending the simply-typed λ-calculus with primitives for thread creation and inter-thread communication. Threads can communicate in two ways: by π-calculus-style synchronous channels, or by waiting for a thread to terminate.

$\mu\nu$CML contains many of the features of Ferreira, Hennessy and Jeffrey's [6] μCML, but is missing the `event` type and its associated functions. We believe that adding the `event` type back into $\mu\nu$CML would pose few technical problems.

The grammar for $\mu\nu$CML types is obtained by extending that for simply-typed lambda calculus with type constructors for thread identifiers and channel identifiers. We assume a grammar B for base types, including at least the unit type `unit`

$$\begin{array}{rcll}
\mathbf{0} \parallel C & \equiv & C & \\
(C_1 \parallel C_2) \parallel C_3 & \equiv & C_1 \parallel (C_2 \parallel C_3) & \\
C_1 \parallel C_2 & \equiv & C_2 \parallel C_1 & \\
C_1 \parallel \nu n\,.\,C_2 & \equiv & \nu n\,.\,(C_1 \parallel C_2) & \quad (n \notin C_1)
\end{array}$$

Figure 3: Axioms for structural congruence $C \equiv C'$

and the boolean type bool. The grammar of types is given:

$$\sigma ::= B \mid \sigma * \sigma \mid \sigma \to \sigma \mid \sigma\,\mathsf{chan} \mid \sigma\,\mathsf{thread}$$

Since we are using a call-by-value reduction semantics, we need a grammar for values. We assume an infinite set of variables x and names n, and some base values b including at least $()$, true and false. The grammar of $\mu\nu$CML values is given:

$$v ::= b \mid (v,v) \mid \lambda x : \sigma\,.\,t \mid n \mid x$$

A thread $\mathsf{let}\,x_1 = e_1\,\mathsf{in} \cdots \mathsf{let}\,x_n = e_n\,\mathsf{in}\,v$ consists of a stack of expressions $e_1,\ldots,e_n$ to be evaluated, followed by a return value v. The grammar of $\mu\nu$CML threads is given:

$$t ::= v \mid \mathsf{let}\,x = e\,\mathsf{in}\,t$$

An expression consists of the usual simply-typed λ-calculus with booleans, together with primitives for multi-threaded computation:

- $\mathsf{chan}\,()$ creates a new channel identifier.
- $\mathsf{send}\,(c,v)$ sends value v along channel c to a matching expression $\mathsf{recv}\,c$, which returns v.
- $\mathsf{spawn}\,v$ creates a new named thread, which executes $v()$, and returns the thread identifier.
- $\mathsf{join}\,i$ blocks waiting for the thread with identifier i to terminate with value v, which is then returned (this is similar to Reppy's [23] *joinVal* function).

The grammar of $\mu\nu$CML expressions is given:

$$\begin{array}{rl} e ::= & t \mid \mathsf{fst}\,v \mid \mathsf{snd}\,v \mid v\,v \mid \mathsf{if}\,v\,\mathsf{then}\,t\,\mathsf{else}\,t \mid v = v \mid \\ & \mathsf{send}\,v \mid \mathsf{recv}\,v \mid \mathsf{chan}\,v \mid \mathsf{join}\,v \mid \mathsf{spawn}\,v \end{array}$$

The type inference rules for threads are given in Figure 1. The type judgements are of the form:

$$\Gamma;\Delta \vdash t : \sigma$$

where Γ is the type context for free variables and Δ the type context for free names.

In order to present the reduction semantics for $\mu\nu$CML it will be useful to describe the configurations of evaluation:

$$C ::= \mathbf{0} \mid C \parallel C \mid \nu n : \sigma\,.\,C \mid n[t]$$

Let the thread names of a configuration be defined:

$$\begin{array}{rclcrcl}
tn(\mathbf{0}) & = & \emptyset & \quad & tn(C_1 \parallel C_2) & = & tn(C_1) \cup tn(C_2) \\
tn(n[t]) & = & \{n\} & & tn(\nu n\,.\,C) & = & tn(C) \setminus \{n\}
\end{array}$$

We require configurations to be linear in their use of thread names: in any configuration $C_1 \parallel C_2$, the thread names of C_1 and C_2 are disjoint, and in any configuration $\nu n : \sigma\,\mathsf{thread}\,.\,C$, n is a thread name of C.

As we can see, a configuration is simply a collection of named threads running concurrently, with shared private names.

We present the type inference rules for configurations in Figure 2. Judgements are of the form:

$$\Delta \vdash C$$

There is an evident structural congruence on configurations given in Figure 3 which should be familiar to readers from the π-calculus [17].

Let the reduction relation $C \to C'$ be the precongruence which includes structural equivalence and the axioms in Figure 4. We have split the reduction rules into confluent rules $C \xrightarrow{\beta} C'$ and one non-confluent communication rule $C \xrightarrow{\tau} C'$. Let $\Rightarrow$ be the reflexive transitive closure of $\to$.

2.1 Observational equivalence, $\approx^b$

We present a notion of observational equivalence for our language following [13], which is a variant of the barbed bisimulation equivalence proposed in [19]. The interested reader can see [8] for a discussion regarding the two approaches.

A binary relation $\mathcal{R}$ on configurations is *contextual* if it satisfies:

$$C_1 \,\mathcal{R}\, C_2 \quad \text{implies} \quad \forall \mathcal{C}\,.\,\mathcal{C}[C_1] \,\mathcal{R}\, \mathcal{C}[C_2]$$

$\mathcal{R}$ is *barbed* if it satisfies:

$$C_1 \,\mathcal{R}\, C_2 \quad \text{implies} \quad \forall n\,.\,C_1 \Downarrow_n \text{ iff } C_2 \Downarrow_n$$

$$
\begin{array}{rcll}
n[\mathsf{let}\,x = v\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[t[v/x]] & \\
n[\mathsf{let}\,x_2 = (\mathsf{let}\,x_1 = e_1\,\mathsf{in}\,t_2)\,\mathsf{in}\,t_3] & \xrightarrow{\beta} & n[\mathsf{let}\,x_1 = e_1\,\mathsf{in}\,(\mathsf{let}\,x_2 = t_2\,\mathsf{in}\,t_3)] & \\
n[\mathsf{let}\,x = \mathsf{fst}\,(v_1, v_2)\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[\mathsf{let}\,x = v_1\,\mathsf{in}\,t] & \\
n[\mathsf{let}\,x = \mathsf{snd}\,(v_1, v_2)\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[\mathsf{let}\,x = v_2\,\mathsf{in}\,t] & \\
n[\mathsf{let}\,x = (\lambda x_1\,.\,t_1)\,v_1\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[\mathsf{let}\,x = (\mathsf{let}\,x_1 = v_1\,\mathsf{in}\,t_1)\,\mathsf{in}\,t] & \\
n[\mathsf{let}\,x = (\mathsf{if}\,\mathsf{true}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[\mathsf{let}\,x = t_1\,\mathsf{in}\,t] & \\
n[\mathsf{let}\,x = (\mathsf{if}\,\mathsf{false}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[\mathsf{let}\,x = t_2\,\mathsf{in}\,t] & \\
n[\mathsf{let}\,x = (v = v)\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[\mathsf{let}\,x = \mathsf{true}\,\mathsf{in}\,t] & \\
n[\mathsf{let}\,x = (v_1 = v_2)\,\mathsf{in}\,t] & \xrightarrow{\beta} & n[\mathsf{let}\,x = \mathsf{false}\,\mathsf{in}\,t] & (v_1 \neq v_2) \\
n[\mathsf{let}\,x = \mathsf{chan}\,()\,\mathsf{in}\,t] & \xrightarrow{\beta} & \nu n'\,.\,n[\mathsf{let}\,x = n'\,\mathsf{in}\,t] & (n' \text{ fresh}) \\
n_1[\mathsf{let}\,x = \mathsf{join}\,n_2\,\mathsf{in}\,t] \parallel n_2[v] & \xrightarrow{\beta} & n_1[\mathsf{let}\,x = v\,\mathsf{in}\,t] \parallel n_2[v] & \\
n[\mathsf{let}\,x = \mathsf{spawn}\,v\,\mathsf{in}\,t] & \xrightarrow{\beta} & \nu n'\,.\,n[\mathsf{let}\,x = n'\,\mathsf{in}\,t] \parallel n'[v\,()] & (n' \text{ fresh}) \\
\\
n_1[\mathsf{let}\,x_1 = \mathsf{send}\,(n, v)\,\mathsf{in}\,t_1] \parallel n_2[\mathsf{let}\,x_2 = \mathsf{recv}\,n\,\mathsf{in}\,t_2] & \xrightarrow{\tau} & n_1[\mathsf{let}\,x_1 = ()\,\mathsf{in}\,t_1] \parallel n_2[\mathsf{let}\,x_2 = v\,\mathsf{in}\,t_2] &
\end{array}
$$

Figure 4: Axioms for reduction precongruence $C \to C'$

where:

$$C \Downarrow_n \quad \text{iff} \quad \exists C'\,.\,C \Rightarrow n[\mathsf{true}] \parallel C'$$

$\mathcal{R}$ is *reduction closed* if it satisfies:

$$\begin{tikzcd} C_1 \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d] & C_2 \\ C_1' & \end{tikzcd} \quad \text{can be completed} \quad \begin{tikzcd} C_1 \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d] & C_2 \arrow[d, Rightarrow] \\ C_1' \arrow[r, leftrightarrow, "\mathcal{R}"] & C_2' \end{tikzcd}$$

and a similar symmetric condition.

Then, *barbed equivalence*, $\approx^b$ is defined to be the largest reduction-closed barbed contextual relation.

It is routine to show that barbed equivalence is a congruence, since we have required it to be contextual.

Barbed equivalence is a natural definition for a bisimulation-like equivalence, but it is very difficult to reason about, since its definition includes a quantification over all contexts. In the remainder of this paper, we shall present a labelled transition system semantics for $\mu\nu$CML and a coinductive presentation of barbed equivalence based on this.

3 Operational semantics and bisimulation equivalence

We make our first steps towards characterizing barbed equivalence using a labelled transition system semantics. We adopt the approach we advocated in [15] by designing a semantics such that:

- Bisimulation can be defined in the standard way (following Gordon [10] and Bernstein's [1] approach to bisimulation for higher-order languages, rather than the higher-order bisimulation used by Thomsen [30] and Ferreira, Hennessy and Jeffrey [6]).
- Labels are *contextual* in the sense that each labelled transition represents a small program fragment which induces an appropriate reduction. This notion of contextual label has been investigated in depth by Sewell [28].

Our labelled transition system is defined as a relation between well-typed configurations. The rules are presented in Figure 5. In addition to these transitions with labels ranged over by γ, we include both β- and τ-reductions of the previous section.

Let α range over γ, τ and β transitions. We define $C \stackrel{\alpha}{\Longrightarrow} C'$ as $C \Rightarrow \xrightarrow{\alpha} \Rightarrow C'$ and $C \stackrel{\hat{\alpha}}{\Longrightarrow} C'$ as $C \Rightarrow C'$ when α is β or τ and $C \stackrel{\alpha}{\Longrightarrow} C'$ otherwise.

The labels used take various forms, many are prepended with an identifier, for example, $\xrightarrow{n.b}$. This signifies which named thread we are currently investigating. Some are followed by another identifier, for example, $\xrightarrow{n.\mathsf{fst}.n'}$ indicates

$$
\begin{array}{rcll}
(\Delta \vdash n[b]) & \xrightarrow{n.b} & (\Delta \vdash n[b]) & \\
(\Delta \vdash n[n'']) & \xrightarrow{n.n''} & (\Delta \vdash n[n'']) & \\
(\Delta \vdash n[(v_1,v_2)]) & \xrightarrow{n.\mathsf{fst}.n'} & (\Delta, n' \vdash n[(v_1,v_2)] \parallel n'[v_1]) & \\
(\Delta \vdash n[(v_1,v_2)]) & \xrightarrow{n.\mathsf{snd}.n'} & (\Delta, n' \vdash n[(v_1,v_2)] \parallel n'[v_2]) & \\
(\Delta \vdash n[\lambda x_1 : \sigma_1 \,.\, t_1]) & \xrightarrow{n.@v_1.n'} & (\Delta, n' \vdash n[\lambda x_1 : \sigma_1 \,.\, t_1] \parallel n'[\mathsf{let}\, x_1 = v_1 \,\mathsf{in}\, t_1]) & (\Delta \vdash v_1 : \sigma_1) \\
(\Delta \vdash n[\mathsf{let}\, x = \mathsf{send}\,(n'', v) \,\mathsf{in}\, t]) & \xrightarrow{\mathsf{send}(n'').n'} & (\Delta, n' \vdash n[\mathsf{let}\, x = ()\,\mathsf{in}\, t] \parallel n'[v]) & \\
(\Delta, n'' : \sigma\,\mathsf{chan} \vdash n[\mathsf{let}\, x = \mathsf{recv}\, n'' \,\mathsf{in}\, t]) & \xrightarrow{\mathsf{recv}(n'',v)} & (\Delta, n'' : \sigma\,\mathsf{chan} \vdash n[\mathsf{let}\, x = v \,\mathsf{in}\, t]) & (\Delta \vdash v : \sigma) \\
(\Delta \vdash \mathbf{0}) & \xrightarrow{\mathsf{join}(v).n} & (\Delta, n : \sigma\,\mathsf{thread} \vdash n[v]) \quad (\Delta \vdash v : \sigma) &
\end{array}
$$

$$
\frac{(\Delta \vdash C_1) \xrightarrow{\gamma} (\Delta' \vdash C_1')}{(\Delta \vdash C_1 \parallel C_2) \xrightarrow{\gamma} (\Delta' \vdash C_1' \parallel C_2)}
\qquad
\frac{(\Delta \vdash C_2) \xrightarrow{\gamma} (\Delta' \vdash C_2')}{(\Delta \vdash C_1 \parallel C_2) \xrightarrow{\gamma} (\Delta' \vdash C_1 \parallel C_2')}
$$

$$
\frac{(\Delta, n : \sigma \vdash C) \xrightarrow{\gamma} (\Delta', n : \sigma \vdash C')}{(\Delta \vdash \nu n : \sigma \,.\, C) \xrightarrow{\gamma} (\Delta \vdash \nu n : \sigma \,.\, C')}[n \notin \gamma]
\qquad
\frac{(\Delta, n : \sigma \vdash C) \xrightarrow{n.n'} (\Delta, n : \sigma \vdash C')}{(\Delta \vdash \nu n : \sigma \,.\, C) \xrightarrow{n.\nu n'} (\Delta, n : \sigma \vdash C')}[n \neq n']
$$

Figure 5: Labelled transition system semantics

that we take the first component of the pair located at thread named n and place it in a new thread named n'. The thread under examination is unaffected by the test which obviates the need for copying transitions to allow repeated testing, *c.f.* [15]. The transitions for modelling the communication primitives are not addressed using a thread identifier because the origin of a communication is not an observable property in this language. Similarly, the transition labelled join simply allocates a value to the named thread, irrespective of any term under investigation. It should be clear that such transitions are necessary in order to distinguish, say,

$$n[\mathsf{let}\, x = \mathsf{join}\, n' \,\mathsf{in}\, \mathsf{true}] \not\approx n[\mathsf{let}\, x = \mathsf{join}\, n' \,\mathsf{in}\, \mathsf{false}].$$

However, it is not observable in this language whether a thread is currently waiting on another to terminate. This bears similarity to the situation of the asynchronous π-calculus [3, 12] and the transitions we use are akin to those for *input receptivity* [12]. The use of the free name context allows us to model the static scoping discipline present in CML. The transition rules we use are those of [15] and the side-conditions in the two rules for inferring transitions under $\nu n.$ contexts ensure privacy and freshness respectively.

We can now prove subject reduction for $\mu\nu$CML, for both the reduction semantics and the lts semantics.

Proposition 3.1 *(Subject Reduction)*

(i) If $\Delta \vdash C$ and $C \to C'$ then $\Delta \vdash C'$.

(ii) If $\Delta \vdash C$ and $(\Delta \vdash C) \xrightarrow{\gamma} (\Delta' \vdash C')$ then $\Delta' \vdash C'$

We now make good on our claim that the labelled transitions presented above actually correspond to small, reduction inducing, contexts of the language. Barbs play an important role here as we use them to establish whether a context has successfully induced a reduction. We list below the actual contexts used for each label. We write C^{Δ}_{γ} to mean the context corresponding to the label γ used on a term with free names in Δ with a barb indicating success at a special fresh location l.

$$
\begin{array}{rcl}
C^{\Delta}_{n.b} & = & l[b = \mathsf{join}\, n] \\
C^{\Delta}_{n.n'} & = & l[n' = \mathsf{join}\, n] \\
C^{\Delta}_{n.\nu n'} & = & l[(\mathsf{join}\, n) \notin \Delta] \\
C^{\Delta}_{n.\mathsf{fst}.n'} & = & l[\mathsf{join}\, n; \mathsf{true}] \parallel n'[\mathsf{fst}\,(\mathsf{join}\, n)] \\
C^{\Delta}_{n.\mathsf{snd}.n'} & = & l[\mathsf{join}\, n; \mathsf{true}] \parallel n'[\mathsf{snd}\,(\mathsf{join}\, n)] \\
C^{\Delta}_{n.@v.n'} & = & l[\mathsf{join}\, n; \mathsf{true}] \parallel n'[\mathsf{join}\, n\, v] \\
C^{\Delta}_{\mathsf{recv}(n,v)} & = & l[\mathsf{send}\,(n,v); \mathsf{true}] \\
C^{\Delta}_{\mathsf{send}(n).n'} & = & l[\mathsf{join}\, n'; \mathsf{true}] \parallel n'[\mathsf{recv}\, n] \\
C^{\Delta}_{\mathsf{join}(v).n} & = & l[\mathsf{true}] \parallel n[v]
\end{array}
$$

where we use obvious syntax sugar such as $n \notin \Delta$.

Proposition 3.2 *(Contextuality)*

(i) If $(\Delta \vdash C) \xrightarrow{\gamma} (\Delta, \Delta' \vdash C')$ then $\Delta \vdash C^{\Delta}_{\gamma} \parallel C \Rightarrow \Delta \vdash l[\mathsf{true}] \parallel \nu\Delta' \,.\, C'$

(ii) If $C^{\Delta}_{\gamma} \parallel C \Rightarrow l[\mathsf{true}] \parallel C'$
then $(\Delta \vdash C) \xRightarrow{\gamma} (\Delta, \Delta' \vdash C'')$
and $C' \xrightarrow{\beta}{}^{*} \nu\Delta' . C''$.

A *type-indexed relation* $\mathcal{R}$ is a family of relations $\mathcal{R}_{\Delta}$ on typed configurations $\Delta \vdash C$. We will often write $\Delta \vDash C_1 \mathrel{\mathcal{R}} C_2$ whenever $(\Delta \vdash C_1) \mathrel{\mathcal{R}_{\Delta}} (\Delta \vdash C_2)$.

A *simulation* is a type-indexed relation on configurations $\mathcal{R}$ such that the following diagram can be completed:

$$\begin{tikzcd}
(\Delta \vdash C_1) \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d, "\alpha"'] & (\Delta \vdash C_2) & \text{as} & (\Delta \vdash C_1) \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d, "\alpha"'] & (\Delta \vdash C_2) \arrow[d, Rightarrow, "\hat{\alpha}"'] \\
(\Delta' \vdash C_1') & & & (\Delta' \vdash C_1') \arrow[r, leftrightarrow, "\mathcal{R}"] & (\Delta \vdash C_2')
\end{tikzcd}$$

A *bisimulation* is a simulation whose inverse is also a simulation. Let $\approx$ denote the largest bisimulation between configurations.

We must allow a certain amount of weakening in the name context in order for bisimulation to be sound. This can be expressed as an open extension. That is:

$$\Delta \vDash C_1 \approx^{\circ} C_2 \quad \text{iff} \quad \Delta, \Delta' \vDash C_1 \approx C_2 \text{ for all fresh } \Delta'$$

On closed expressions (including values and threads) we can define bisimulation to be given by placing the expression in a configuration:

$$\Delta \vDash e_1 \approx e_2 : \sigma \quad \text{iff} \quad \Delta, n : \sigma\,\mathsf{thread} \vDash n[e_1] \approx n[e_2]$$

For open expressions, we define the open extension as usual, by substituting through with appropriately typed values (allowing ourselves to introduce new names while doing so):

$$\begin{aligned} &\overline{x} : \overline{\sigma}; \Delta \vDash e_1 \approx^{\circ} e_2 : \sigma \\ &\quad \text{iff} \quad \Delta, \Delta' \vDash e_1[\overline{v}/\overline{x}] \approx e_2[\overline{v}/\overline{x}] \text{ for all } \Delta, \Delta' \vdash \overline{v} : \overline{\sigma} \end{aligned}$$

We will now make some observations about β-reduction which will prove useful in the remainder of the paper. It is not too hard to see that the reductions we identified as being β-reductions are in fact confluent. They are not only confluent with respect to other reductions, but in fact with respect to labelled transitions:

Proposition 3.3 *The follow diagram can be completed:*

$$\begin{tikzcd}
(\Delta \vdash C) \arrow[r, "\beta"] \arrow[d, "\alpha"'] & (\Delta \vdash C') & \textit{as} & (\Delta \vdash C) \arrow[r, "\beta"] \arrow[d, "\alpha"'] & (\Delta \vdash C') \arrow[d, "\alpha"'] \\
(\Delta' \vdash C'') & & & (\Delta' \vdash C'') \arrow[r, "\beta"] & (\Delta' \vdash C''')
\end{tikzcd}$$

or $C' \equiv C''$ *if* α *is* β.

This observation now allows us to state a proof principle which we utilize heavily, namely weak bisimulation up to β-reduction. We say that a type-indexed relation $\mathcal{R}$ is a simulation up to $(\xrightarrow{\beta}{}^{*}, \approx)$ if we can complete the diagram:

$$\begin{tikzcd}
(\Delta \vdash C_1) \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d, "\alpha"'] & (\Delta \vdash C_2) & \text{as} & (\Delta \vdash C_1) \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d, "\alpha"'] & (\Delta \vdash C_2) \arrow[d, Rightarrow, "\hat{\alpha}"'] \\
(\Delta' \vdash C_1') & & & (\Delta' \vdash C_1') \arrow[r, leftrightarrow, "\xrightarrow{\beta}{}^{*}\mathcal{R}\approx"] & (\Delta' \vdash C_2')
\end{tikzcd}$$

As before we say that $\mathcal{R}$ is a bisimulation up to $(\xrightarrow{\beta}{}^{*}, \approx)$ if both $\mathcal{R}$ and its inverse are simulations up to $(\xrightarrow{\beta}{}^{*}, \approx)$. The proof principle we appeal to is

Proposition 3.4 *If* $\mathcal{R}$ *is a bisimulation up to* $(\xrightarrow{\beta}{}^{*}, \approx)$ *then* $\approx\mathcal{R}\approx$ *is a bisimulation.*

Proof: We use the fact that $\mathcal{R}$ is a bisimulation up to $(\xrightarrow{\beta}{}^{*}, \approx)$ and confluence of β-reduction, Proposition 3.3, to show that we can complete the diagram:

$$\begin{tikzcd}
(\Delta \vdash C_1) \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d, Rightarrow, "\hat{\alpha}"'] & (\Delta \vdash C_2) & \text{as} & (\Delta \vdash C_1) \arrow[r, leftrightarrow, "\mathcal{R}"] \arrow[d, Rightarrow, "\hat{\alpha}"'] & (\Delta \vdash C_2) \arrow[d, Rightarrow, "\hat{\alpha}"'] \\
(\Delta' \vdash C_1') & & & (\Delta' \vdash C_1') \arrow[r, leftrightarrow, "\xrightarrow{\beta}{}^{*}\mathcal{R}\approx"] & (\Delta' \vdash C_2')
\end{tikzcd}$$

Given this it is straightforward to demonstrate that $\approx\mathcal{R}\approx$ is a bisimulation by, again using confluence, observing that $\xrightarrow{\beta}{}^{*}$ is contained in $\approx$. □

We will now show that bisimulation for $\mu\nu$CML coincides with barbed congruence. Soundness follows immediately once we have that bisimulation is a congruence: this is the subject of the next section.

Proposition 3.5 $\Delta \vDash C \approx^{\circ} C'$ *implies* $\Delta \vDash C \approx^{b} C'$

Proof: It is easy to show that $\approx$ is barbed and reduction-closed, and Theorem 4.7 shows that $\approx^{\circ}$ is a congruence. Hence, $\approx^{\circ}$ implies $\approx^{b}$. □

Completeness is a simple corollary of contextuality, Proposition 3.2.

Proposition 3.6 $\Delta \vDash C \approx^{b} C'$ *implies* $\Delta \vDash C \approx^{\circ} C'$

Proof: (Outline) We aim to show that $\approx^{b}$ is a bisimulation up to $(=, \approx)$. Suppose that $C_1 \approx^{b} C_2$ and suppose that $C_1 \xrightarrow{\alpha} C_1'$. Choose some $l \notin C_1, \alpha$. We note by Proposition 3.2, that $C_{\alpha} \parallel C_1 \Rightarrow l[\mathsf{true}] \parallel C_1'$ and use the fact that

$\approx^b$ is a congruence, reduction closed and barbed to establish that $C_\alpha \parallel C_2 \Rightarrow l[\mathsf{true}] \parallel C_2'$ for some C_2' such that $l[\mathsf{true}] \parallel C_1' \approx^b l[\mathsf{true}] \parallel C_2'$. It is easy to show from here that $C_1' \approx^b C_2'$ as $l \notin C_1', C_2'$ and use Proposition 3.2 again to see that $C_2 \overset{\hat{\alpha}}{\Longrightarrow} \approx C_2'$. More care must be taken in the case where α is $n \,.\, \nu n''$ but a similar argument can be applied. We then use Proposition 3.4 to conclude. □

4 Congruence properties of bisimulation

We are left with the task of showing that bisimulation is a congruence. This is a notoriously difficult problem, and proof techniques which work in the presence of both higher-order features and unique names are limited [21, 22].

A viable approach to tackling this problem in languages with sufficient power is to represent higher-order computation by first-order means. Indeed, Sangiorgi demonstrates in his thesis [26] that higher-order π-calculus can be encoded, fully abstractly, in the first-order π-calculus by means of reference passing—this transformation is described in two stages, the first of which is technically known as a trigger encoding and recasts higher-order π-calculus in a sublanguage of itself in which only canonical higher-order values, or triggers, are passed.

We adopt a similar approach here but, owing to the functional nature of the language we find that our encoding to be more complicated than that of the higher-order π-calculus. The thrust of the current work is to demonstrate a novel approach to proving a fully abstract trigger encoding which can be used to prove congruence of bisimulation in higher-order languages.

Rather than compositionally translating our higher-order language into a simpler language, we describe an alternative operational semantics which implements this trigger passing. The intention is that there is a direct proof of congruence of bisimulation equivalence on this alternative operational semantics, and correctness between the two semantics yields congruence on the original. Correctness between the two semantics can be stated quite tightly as:

$$[\![C]\!]_\omega \approx [\![C]\!]_0$$

where $[\![C]\!]_\omega$ is understood to be the interpretation of C using the original semantics and $[\![C]\!]_0$ the triggered semantics.

In fact, to relieve the difficulty of proving correctness we aim to use an induction on the order of the type of C. This leads us to defining a hierarchy of semantics, indexed by type order. In $[\![C]\!]_n$, terms of type higher than n are passed directly, and terms of lower type are trigger-encoded. We can then regard $[\![C]\!]_\omega$ as the 'limit' of the semantics $[\![C]\!]_0, [\![C]\!]_1, \ldots$. Our proof that bisimulation is a congruence is then broken into three parts:

1. Prove that bisimulation is a congruence for $[\![\cdot]\!]_0$.
2. Prove that if $\forall i \,.\, [\![C]\!]_0 \approx [\![C']\!]_i$ then $[\![C]\!]_0 \approx [\![C']\!]_\omega$.
3. Prove that $[\![C]\!]_i \approx [\![C]\!]_{i+1}$.

From these three properties, it is easy to prove that bisimulation is a congruence for $[\![\cdot]\!]_\omega$, which is, by definition, exactly our original semantics for $\mu\nu$CML.

Note that this proof relies on a well-founded order on types, and so will not work in the presence of general recursive types. This is not as limiting as might first be thought, since $\sigma\,\mathsf{chan}$ and $\sigma\,\mathsf{thread}$ are considered to be order 0 no matter the order of σ, and so we can deal with any recursive type as long as the recursive type variable is beneath a $\cdot\,\mathsf{chan}$ or $\cdot\,\mathsf{thread}$. This is a similar situation as for most imperative languages, which restrict recursive types to those including pointers. Also note that this restriction is weak enough to include all of the π-calculus sorts, such as the type $\mu X \,.\, X\,\mathsf{chan}$ which describes monomorphic π-calculus channels.

We will now present the triggered semantics and show the three required properties.

4.1 Trigger Semantics for $\mu\nu$CML

In order to describe these semantics concisely it will be helpful to introduce a mild language extension. There is no explicit recursion function definitions in the core language we presented above as such terms can be programmed up using the thread synchronization primitives (*c.f.* coding the Y-combinator using general references). We introduce a replicated reception primitive, which can indeed be coded using recursive functions. Let us write $*\mathsf{recv}\, n$ to represent this new expression. There is an associated reduction rule for this new expression which behaves as a recv expression but spawns a new thread of evaluation. This is defined as

$$\begin{array}{c} n_1[\mathsf{let}\, x_1 = \mathsf{send}\,(n,v)\,\mathsf{in}\, t_1] \parallel n_2[\mathsf{let}\, x_2 = *\mathsf{recv}\, n\,\mathsf{in}\, t_2] \\ \rightarrow \\ \nu n_3 \,.\, \left(\begin{array}{c} n_1[\mathsf{let}\, x_1 = ()\,\mathsf{in}\, t_1] \parallel n_2[\mathsf{let}\, x_2 = v\,\mathsf{in}\, t_2] \parallel \\ n_3[\mathsf{let}\, x_2 = *\mathsf{recv}\, n\,\mathsf{in}\, t_2] \end{array} \right) \end{array}$$

Of course, there is an obvious corresponding transition rule for replicated reception also. The following pieces of notation will be convenient. Let τ_a denote the term

$$\lambda x \,.\, \mathsf{let}\, r = \mathsf{chan}\,()\,\mathsf{in}\,\mathsf{send}\,(a,(x,r)); \mathsf{recv}\, r$$

The *trigger call* τ_a is used to substitute through terms in place of functions. When the trigger call is applied to an argument, the trigger simply sends the argument off to the actual function (on channel a). It must also wait for the resulting value given by the application on a freshly created private channel.

Complementary to this is the *resource* at a, written $a \Leftarrow f$, where we use f to range over λ-abstractions. This is defined to be a replicated receive command:

$$\mathsf{let}\, x_1, x_2 = *\mathsf{recv}\, a\, \mathsf{in}\, \mathsf{let}\, z = f\, x_1\, \mathsf{in}\, \mathsf{send}\, (x_2, z)$$

which can continually receive arguments to f, along with a reply channel. It then applies f to the argument and sends the result back along the reply channel.

These are the two basic components of the triggered semantics. We use them to define a notion of type-indexed substitution. Recall that the order of a type $O(\sigma)$ is defined by induction such that $O(\sigma_1) < O(\sigma_1 \to \sigma_2)$ and $O(\sigma\,\mathsf{thread}) = O(\sigma\,\mathsf{chan}) = 0$ and the type-order of a term $O(t)$ is the order of the terms type. Let the level i substitution $[v/x]_i$ be defined by:

$$\begin{array}{lcl} C[b/x]_i & = & C[b/x] \\ C[n/x]_i & = & C[n/x] \\ C[(v_1,v_2)/x]_i & = & (C[(x_1,x_2)/x])[v_1/x_1]_i[v_2/x_2]_i \\ C[f/x]_i & = & \left\{ \begin{array}{l} C[f/x] \text{ if } O(f) \leq i \text{ or } f = \tau_a \\ \nu a, n\,.\,(C[\tau_a/x] \parallel n[a \Leftarrow f]) \text{ otherwise} \end{array} \right. \end{array}$$

¿From the definition of level i substitution, we can now define the level i trigger semantics $[\![\cdot]\!]_i$ by replacing the β-reduction rule for let expressions with

$$n[\mathsf{let}\, x = v\, \mathsf{in}\, t] \xrightarrow{\beta} n[t[v/x]_i]$$

and leaving all other rules unchanged.

We now state some useful lemmas concerning the trigger protocol semantics.

Lemma 4.1 *Any reductions which are instances of the following are confluent* β*-reductions:*

$$\begin{array}{c} \nu r\,.\,(n[\mathsf{let}\, x = \mathsf{recv}\, r\, \mathsf{in}\, t] \parallel n'[\mathsf{let}\, x' = \mathsf{send}\,(r,v)\, \mathsf{in}\, t']) \\ \xrightarrow{\beta} \\ \nu r\,.\,(n[\mathsf{let}\, x = v\, \mathsf{in}\, t] \parallel n'[\mathsf{let}\, x' = ()\, \mathsf{in}\, t']) \end{array}$$

and

$$\begin{array}{c} \nu n\,.\,(C \parallel n_1[\mathsf{let}\, x_1 = \mathsf{send}\,(n,v)\, \mathsf{in}\, t_1] \parallel n_2[\mathsf{let}\, x_2 = *\mathsf{recv}\, n\, \mathsf{in}\, t_2]) \\ \xrightarrow{\beta} \\ \nu n n_3\,.\left(\begin{array}{c} C \parallel n_1[\mathsf{let}\, x_1 = ()\, \mathsf{in}\, t_1] \parallel \\ n_3[\mathsf{let}\, x_2 = *\mathsf{recv}\, n\, \mathsf{in}\, t_2] \parallel n_2[\mathsf{let}\, x_2 = v\, \mathsf{in}\, t_2] \end{array} \right) \end{array}$$

providing C does not contain a $\mathsf{recv}\, n$ *expression.*

The first of these is observes that channels which are used linearly have unique points of communication and hence give confluent communication. This is used for the return part of the trigger protocol. The latter is slightly more involved and relies upon a side-condition that the sending participant cannot communicate with any party other than the replicated input. We use this property when beginning each trigger protocol communication and can maintain it as an invariant throughout testing.

Lemma 4.2 $[\![C[v/x]_i]\!]_{i'} \approx [\![C[v/x]_{i+1}]\!]_{i'}$ *for all $i' \geq i$.*

This essentially states that, for substitutions of functions of type order i the trigger protocol correctly implements the substitution. In order to see this we show that the relation $\{C[v/x]_i, C[v/x]_{i+1}\}$ is a bisimulation up to $(\xrightarrow{\beta}^*, \approx)$. The difficulty here is seen in the case in which v is a function of order $i+1$, being applied to some argument in C. On the right hand side we have a standard substitution and a standard β-reduction. Whereas on the left hand side we see a triggered substitution, and, by virtue of the argument being of type $< i+1$, a standard β-reduction. It is crucial that no nested trigger substitution is incurred here and we can use the power of the up to technique to finish by appealing to the next lemma, which establishes correctness of the return end of the protocol.

Lemma 4.3

$$\begin{array}{r} [\![C \parallel \nu n' r\,.\,(n[\mathsf{let}\, x = \mathsf{recv}\, r\, \mathsf{in}\, t_2] \parallel n'[\mathsf{let}\, z = t_1\, \mathsf{in}\, \mathsf{send}\,(r,z)])]\!]_i \\ \approx [\![C \parallel n[\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2]]\!]_i \end{array}$$

Again, this lemma is proved using a bisimulation up to $(\xrightarrow{\beta}^*, \approx)$.

Proposition 4.4 *(i) If $[\![C]\!]_0 \approx [\![C']\!]_i$ for all i then $[\![C]\!]_0 \approx [\![C']\!]_\omega$.*

(ii) $[\![C]\!]_i \approx [\![C]\!]_{i+1}$ for all i.

Proof: For Part (i) we construct a bisimulation:

$$\mathcal{R} = \{([\![C]\!]_0, [\![C']\!]_\omega) \mid \exists i\,.\,\forall i' > i\,.\,[\![C]\!]_0 \approx [\![C']\!]_{i'}\}$$

Part (ii) is easy to show using a bisimulation up to $(\equiv, \approx)$ and Lemma 4.2 with i' instantiated to i and $i+1$. □

Corollary 4.5 $[\![C]\!]_0 \approx [\![C]\!]_\omega$ *for all C.*

4.2 Congruence

We have described how, in order to verify congruence of bisimulation equivalence for the standard semantics, it is sufficient to verify congruence of bisimulation equivalence for the completely triggered, level 0, semantics. We show this now.

Proposition 4.6 *For all contexts $\Delta, \Delta' \vdash \mathcal{C}$ of appropriate type* *(i) If* $\Delta \vDash [\![C_1]\!]_0 \approx^\circ [\![C_2]\!]_0$ *then* $\Delta, \Delta' \vDash [\![\mathcal{C}[C_1]]\!]_0 \approx [\![\mathcal{C}[C_2]]\!]_0$

(ii) If $\Gamma; \Delta \vDash [\![e_1]\!]_0 \approx^\circ [\![e_2]\!]_0$ *then* $\Delta, \Delta' \vDash [\![\mathcal{C}[e_1]]\!]_0 \approx [\![\mathcal{C}[e_2]]\!]_0$

Proof: This can now be proved fairly directly using our bisimulation up to technique. The level 0 semantics ensure that the only substitution which occurs is for base values, names and triggers. Bisimulation on these values is just syntactic identity so any problems with *substitutivity* (in the presence of static scoping) which arise in [10, 15] are avoided. □

Given this we can draw upon the results of Corollary 4.5 and the above Proposition to obtain:

Theorem 4.7 *Bisimulation equivalence is a congruence.*

5 Normal bisimulation

So far we have shown that bisimulation equivalence coincides with barbed equivalence. The motivation for providing such a characterisation lies in the need to alleviate the quantification over all contexts present in the definition of barbed equivalence. We achieve this to an extent by reducing contexts to labelled transitions. However, despite being a neater coinductive equivalence, the definition of bisimulation equivalence now quantifies over all transitions. We must question whether this is truly a lighter quantification. One measure we proposed in [15] to answer such a question was to demand that labels be *applicative*. That is to say, whenever a label contains an arbitrary value, the type of that value should be strictly less than the thread being tested. Our labelled transition system defined above is certainly not applicative in this sense. In particular, the $\mathsf{join}(v) \,.\, n$ labels have no such restriction and grant powerful testing abilities. In order to rectify this shortcoming of bisimulation we provide a cut-down lts semantics which do have applicative labels for which bisimulation equivalence coincides with the original. This new semantics is closely related to *normal bisimulation* of Sangiorgi [26], in which a beautifully simple characterisation of bisimulation for higher-order π-calculus is achieved by restricting test values to be either names or trigger calls alone. We adopt the same approach here by defining *canonical values* to be those of the form

$$v_c ::= b \mid (v_c, v_c) \mid \tau_a \mid n \mid x$$

Now, the canonical, or normal semantics for configurations is given by the lts rules in Figures 4, 5 with all values in the transition labels restricted to be canonical. Write $[\![C]\!]_0^c$ to signify the canonical semantics with level 0 substitutions. If we can show that bisimulation equivalence is a congruence on configurations for these semantics it is then a simple step to show

Theorem 5.1 $\Delta \vDash C \approx C'$ *if and only if* $\Delta \vDash [\![C]\!]_0^c \approx [\![C']\!]_0^c$

which justifies the claim that our bisimulation is a simple characterisation of barbed equivalence.

This does however oblige us to show congruence for the canonical semantics.

Proposition 5.2 *If* $\Delta \vDash [\![C_1]\!]_0^c \approx [\![C_2]\!]_0^c$ *then for all contexts* $\Delta, \Delta' \vdash \mathcal{C}$ *we have* $\Delta, \Delta' \vDash [\![\mathcal{C}[C_1]]\!]_0^c \approx [\![\mathcal{C}[C_2]]\!]_0^c$.

Proof: (Outline) There are two operators which must preserve bisimulation equivalence, that is name abstraction and parallel. The former is straightforward but the latter requires some work. We cannot present the full details of this here but try to indicate how one proceeds. For the remainder all configurations are to be understood using the canonical, level 0 semantics. Define:

$$\mathcal{R}^{\Delta} = \left\{ (\nu\bar{k} \,.\, C_1 \parallel C, \nu\bar{k} \,.\, C_2 \parallel C) \mid \Delta, \bar{k} \vDash C_1 \approx C_2 \right\}$$

and show that $\mathcal{R}$ is a bisimulation. In order simplify this we use an up to technique involving strong bisimulation equivalence [16, 20]. In fact we show that $\mathcal{R}$ is a bisimulation up to $(\xrightarrow{\beta}{}^{*}\sim, \approx)$ where $\sim$ denotes strong bisimulation. The reader is invited to check that this is a valid proof technique because confluence implies commutativity of $\xrightarrow{\beta}{}^{*}$ and $\sim$.

We must check the bisimulation closure property. This is straightforward for all cases save for those consisting of an interaction between C_i and C, in particular, communication between these configurations and join synchronizations. For instance, supposing C_1 had a join-call to some thread n_0 in C and C had a value v at that thread. The resulting state after some β-reductions would have C_1 with a trigger call substituted through for the join-call and the resource for v in parallel with C. We can simulate this reduction on C_1 alone using a $\mathsf{join}(\tau_a) \,.\, n_0$ transition. The configuration C_2 must match this to a bisimilar state. The property below guarantees, up to strong bisimulation, that this transition simulates the interaction of C_1 and C sufficiently well:

$$C \parallel n_0[f] \parallel \nu n \,.\, n[a \Leftarrow f] \sim C \parallel n_0[\tau_a] \parallel \nu n \,.\, n[a \Leftarrow f]$$

The remaining cases are analyzed in a similar manner but are quite complicated. The full details highlight the role of the trigger semantics well and will appear in a forthcoming technical report. □

6 Concluding remarks

We have developed an operational account of program equivalence for a fragment of concurrent ML which features higher-order functions, concurrency primitives and statically-scoped local names. The bisimulation equivalence, and in particular that for the canonical semantics provide a lightweight characterisation of barbed equivalence in this setting. This is the first such treatment for a language containing all of these features.

The proof techniques employed here owe much to Sangiorgi [26] and we consider the hierarchical approach to trigger correctness a useful generalization of Sangiorgi's method to the functional setting. Indeed such techniques could be employed in any functional language sufficiently expressive to encode the trigger passing mechanism. We have also identified a useful 'bisimulation up to' technique based on confluent reduction.

There is a striking relationship between location based mobile agent languages in the sense of [5, 24, 29] and the thread identifiers. It could be fruitful to adapt the techniques used here to such a setting. In particular, trigger encodings could address the issues of migrating processes and scope in much the same way they help us achieve congruence here.

References

[1] K.L. Bernstein and E.W. Stark. Operational semantics of a focussing debugger. In *Proc. MFPS*, number 1 in Electronic Notes in Comp. Sci. Springer-Verlag, 1995.

[2] D. Berry, R. Milner, and D.N. Turner. A semantics for ML concurrency primitives. In *Proc. POPL*, 1992.

[3] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, Sophia-Antipolis, 1991.

[4] G. Boudol. The π-calculus in direct style. *Higher-Order and Symbolic Computation*, 11, 1998.

[5] L. Cardelli and A. Gordon. Mobile ambients. In *Proc. FoSSaCS*, Lecture Notes in Comp. Sci. Springer-Verlag, 1998. To appear in *Theoretical Comp. Sci*.

[6] W. Ferreira, M. Hennessy, and A. S. A. Jeffrey. A theory of weak bisimulation for core CML. *J. Functional Programming*, 8(5):447–491, 1998.

[7] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proc. POPL*, 1996.

[8] C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proc. ICALP*, volume 1443 of *Lecture Notes in Comp. Sci.* Springer-Verlag, 1998.

[9] C. Fournet, G. Gonthier, J-J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proc. CONCUR*, volume 1119 of *Lecture notes in computer science*, pages 406–421. Springer-Verlag, 1996.

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

[11] A.D. Gordon and P.D. Hankin. A concurrent object calculus: Reduction and typing. In *Proc. HLCL*, 1998.

[12] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. ECOOP*, 1991.

[13] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Comp. Sci.*, 152(2):437–486, 1995.

[14] D. Howe. Equality in lazy computation systems. In *Proc. LICS*, pages 198–203. IEEE Computer Society Press, 1989.

[15] A.S.A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. In *Proc. LICS*, pages 56–66. IEEE Computer Society Press, 1999.

[16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Comp. Sci.* Springer-Verlag, 1980.

[17] R. Milner. *Communication and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.

[19] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proc. ICALP*, volume 623 of *Lecture Notes in Comp. Sci.* Springer-Verlag, 1992.

[20] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Comp. Sci.*, pages 167–183. Springer-Verlag, 1981.

[21] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. MFCS*, pages 122–141. Springer-Verlag, 1993. Lecture Notes in Comp. Sci. 711.

[22] A.M. Pitts and I.D.B. Stark. Operational reasoning for functions with local state. In A.D. Gordon and A.M. Pitts, editors, *Proc. HOOTS*, pages 227–273. Cambridge University Press, 1998. Publications of the Newton Institute.

[23] J. Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, 1992. Technical Report TR 92-1285.

[24] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proc. POPL*. ACM Press, 1998.

[25] M.Tofte R.Milner and R.Harper. *The Definition of Standard ML*. MIT Press, 1990.

[26] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1993.

[27] D. Sangiorgi and R. Milner. On the problem of 'weak bisimulation up to'. In W.R. Cleaveland, editor, *Proc. CONCUR*, volume 630 of *Lecture Notes in Comp. Sci.*, pages 32–46. Springer-Verlag, 1992.

[28] P. Sewell. From rewrite rules to bisimulation congruences. In D. Sangiorgi and R. de Simone, editors, *Proc. CONCUR*, volume 1466 of *Lecture Notes in Comp. Sci.*, pages 269–284. Springer-Verlag, 1998.

[29] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. ICALP*, number 1443 in Lecture Notes in Comp. Sci., pages 695–706. Springer-Verlag, 1998.

[30] B. Thomsen. *Calculi for Higher-Order Communicating Systems*. PhD thesis, University of London, 1990.