UNIVERSITY OF SUSSEX

COMPUTER SCIENCE

Combining the typed λ-calculus with CCS

W. Ferreira
M. Hennessy
A.S.A. Jeffrey

Report 2/96 May 1996

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

ISSN 1350–3170

Combining the typed λ-calculus with CCS

W. FERREIRA, M. HENNESSY and A.S.A. JEFFREY

ABSTRACT. We investigate a language obtained by extending the typed call-by-value λ calculus with the communication constructs of *CCS*. The language contains two interrelated syntactic classes, *processes* and *expressions*. The former are defined using the *CCS* constructs of choice, parallelism, and action prefixing of *expressions*, where these *expressions* come from a syntactic class which also includes the standard constructs from the call-by-value λ-calculus.

We define a higher order bisimulation equivalence and prove that it is a congruence for *expressions*; when modified in the standard manner to take into account initial τ moves it is also a congruence for *processes*. We then show that when applied to *expressions* this semantic theory is a generalisation of the theory of equality for the call-by-value λ calculus while when applied to *processes* it is an extension of the theory of bisimulation congruence of *CCS*.

1 Introduction

CCS is an abstract process description language whose study and understanding, [7], has been of great signifi cance in the development of the theory of concurrency. An algebraic view is taken of processes in that their description is in terms of a small collection of primitive *constructors*, such as choice $+$, parallelism $||$ and action prefi xing $a?, a!$. These action prefi xes designate the sending and receiving of a synchronisation impulse along a virtual channel a. Communication is deemed to be the simultaneous occurrence of these two events and is denoted by the special action τ. So *CCS* expressions describe processes in terms of their synchronisation or communication potentials and the algebraic theory, expressed as equations over the constructors, is validated in terms of behavioural equivalences defi ned using the these potentials.

Much research has been carried out on extending this elegant theory to more expressive process descriptive languages. Here we are concerned with languages in which the synchronisation is replaced by the exchange of data, where the abstract actions $a?$ and $a!$ are instantiated to $a?x$ and $a!v$, the reception and sending of data. In papers such as [5, 9], and even in [7], such extensions are considered but the domain of transmittable values is taken to have no computational signifi cance. All data expressions denote a unique value and the computation of this value is not of concern. Here we are interested in situation in which the data space may be computationally complex and their evaluation may effect the behaviour of processes which use them.

This work was partially supported by the EU EXPRESS Working Group and the Royal Society.

A typed λ-calculus, based on some primitive set of data types, provides a non-trivial example of such a data space. It is also a very useful example as there are various existing programming languages, such as *CML* [10], *Facile* [3], which are based on different methods for combining the communication primitives of *CCS* with the typed λ-calculus.

In this paper we try, where possible, to unify *CCS* directly with the typed call-by-value λ-calculus, to find a *communicate-by-value* concurrent language. However, it is not possible to fully unify the process language with the functional language, due to the behaviour of *CCS* summation. The operational semantics for β-reduction includes:

$$(\lambda().e)() \xrightarrow{\tau} e$$

If we were to allow function applications in $+$ contexts we would therefore have:

$$a!1 + (\lambda().b!1)() \xrightarrow{\tau} b!1 \qquad a!1 + b!1 \not\Rightarrow b!1$$

and so we would have:

$$(\lambda().e)() \neq e$$

We consider this to be an undesirable property of a functional language, and so we will distinguish between *processes* (which may be placed in $+$ contexts) and *expressions* (which may not). For expressions, weak bisimulation is a congruence, and we have:

$$(\lambda().e)() \approx e$$

For processes, we have to use observational equivalence, and we have:

$$\tau.p \neq p$$

In this paper we distinguish syntactically between processes and expressions: in languages such as *CML* and *Facile* this distinction is made by the type system.

In Section 2 we describe a language λ_v^{con} which combines the call-by-value λ-calculus expressions with communicate-by-value *CCS* processes.

In Section 3 we then develop a variation of weak bisimulation equivalence, based on *higher-order bisimulations*. As is to be expected the resulting equivalence is not preserved by choice contexts. However when it is adjusted in the standard manner, [7], the resulting *higher order bisimulation congruence* is preserved by all λ_v^{con} contexts. We also show the resulting theory is both a generalisation of the standard theory of *CCS* and call-by-value λ-calculus, and that all closed expressions can be converted to head normal form.

In Section 4 we discuss possible extensions, such as the use of symbolic techniques, and the call-by-name variant of the language.

2 The Language

This section is divided into three parts. The first outlines the syntax and operational semantics of a call-by-value λ-calculus, the second provides a communicate-by-value process language based on *CCS* [7], and the third combines the languages together as far as possible.

2.1 A call-by-value λ-calculus

The language we use is typed, the types been given by the grammar:

$$A ::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid A \to A$$

Here int is used as an example of a *basic* type and we can easily incorporate others. For each type A we assume an infinite set of variables, Var_A, ranged over by x_A, and some (indefinite) set of constants for manipulating values of the basic types, a typical example being succ the *successor* function over int. However to increase readability we will omit the typing information from variables and constants unless absolutely necessary.

The syntax of the language is given by the following grammar:

$$\begin{aligned} e, f \ldots \in Exp &::= v \mid c\,e \mid \mathsf{if}\,e\,\mathsf{then}\,e\,\mathsf{else}\,e \mid \mathsf{let}\,x_A{=}e\,\mathsf{in}\,e \mid e\,e \\ v, w, \ldots \in Val &::= l \mid \mu x_{A\to B}.(\lambda y_A.e) \mid x_A \\ l \in Lit &::= \mathsf{true} \mid \mathsf{false} \mid () \mid 0 \mid 1 \mid \ldots \end{aligned}$$

The main syntactic category is *Exp* but a sub-category of *value* expressions is also defined. The only non-trivial value expressions are function abstractions, or more accurately recursively defined function abstractions. Informally $\mu x.(\lambda y.e)$ denotes a recursively defined function whose definition could also be rendered as:

$$x \Longleftarrow \lambda y.e$$

If x does not occur in e then we will often abbreviate $\mu x.(\lambda y.e)$ to $\lambda y.e$.

Function abstraction and let are the only variable binding construct in the language. This leads to the standard definition of free and bound occurrences of variables, open and closed expressions, α-equivalence, $=_\alpha$, and of substitution. However we only ever require substitutions of the form $e[v/x]$, i.e. the substitution of a value v for all free occurrences of a variable x in an expression e.

We can associate with every expression at most one type. The typing judgements have a very simple form $\vdash e : A$ and the rules for inferring types are given in Figure 1. Of course we also need inference rules for the literals of any extra basic types used. We use $\lambda_v, (\lambda_v{}^\circ)$ to denote the set of all closed (open) well-typed expressions.

Intuitively every expression of type A should denote a value of this type and the operational semantics describes, in an abstract form, a procedure for evalu-

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

$$\frac{}{\vdash () : \mathsf{unit}} \qquad \frac{}{\vdash n : \mathsf{int}}[n \in \{0,1,\ldots\}]$$

$$\frac{\vdash e : A \quad \vdash f : B}{\vdash \mathsf{let}\, x_A = e \,\mathsf{in}\, f : B} \qquad \frac{\vdash e : B}{\vdash \mu x_{A\to B}.(\lambda y_A.e) : A \to B}$$

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

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

FIGURE 1. Type Rules for λ expressions.

ating expressions to values in a call-by-value manner; of course because of the presence of recursion this evaluating procedure may never terminate. Formally the operational semantics is given in terms of two relations between closed expressions. The first, $e \xrightarrow{\tau}_e e'$, means that in one reduction step the evaluation of e can be reduced to that of e'. For example to evaluate the expression $\mathsf{let}\, x = e \,\mathsf{in}\, f$ we first evaluate e, using the rule:

$$\frac{e \xrightarrow{\tau}_e e'}{\mathsf{let}\, x = e \,\mathsf{in}\, f \xrightarrow{\tau}_e \mathsf{let}\, x = e' \,\mathsf{in}\, f}$$

When e produces a value v then the evaluation proceeds evaluating the expression $f[v/e]$. In order to express this formally we need a second relation, $e \xrightarrow{\surd v}_e$ which tells when an expression e has produced a value v (the reason for the non-standard notation will become clear when we unify the expression and process languages in Section 2.3). The evaluation of $\mathsf{let}\, x = \ldots \,\mathsf{in}\, \ldots$ expressions is captured by the additional rule:

$$\frac{e \xrightarrow{\surd v}_e}{\mathsf{let}\, x = e \,\mathsf{in}\, f \xrightarrow{\tau}_e f[v/x]}$$

Similarly to evaluate an application $e\, f$, the expression e is evaluated until we reach a value $e \xrightarrow{\tau}_e^* \xrightarrow{\surd v}_e$. Since e is well-typed this value v must be of the form $\mu x.(\lambda y.g)$ and the evaluation proceeds by evaluating $\mathsf{let}\, y = f \,\mathsf{in}\, g[v/x]$; this is captured by the rule:

$$\frac{e \xrightarrow{\surd \mu x.(\lambda y.g)}_e}{e\, f \xrightarrow{\tau}_e \mathsf{let}\, y = f \,\mathsf{in}\, g[\mu x.(\lambda y.g)/x]}$$

The application of constants $c\,e$ is handled in a similar manner; e is evaluated to a value v and then the value produced by $c\,v$ depends on the constant in question. The effect of constants can be expressed in terms of a function δ, which given a constant and a value returns the corresponding expression. For example the behaviour of the constant succ is given by $\delta(\mathsf{succ}, n) = n + 1$.

Values:

$$\frac{}{v \xrightarrow{\surd v}_e}$$

Reductions:

$$\frac{e \xrightarrow{\surd \mu x.(\lambda y.g)}_e}{e\, f \xrightarrow{\tau}_e \mathsf{let}\, y = f \,\mathsf{in}\, g[\mu x.(\lambda y.g)/x]} \qquad \frac{e \xrightarrow{\surd v}_e}{\mathsf{let}\, x = e \,\mathsf{in}\, f \xrightarrow{\tau}_e f[v/x]}$$

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

$$\frac{e \xrightarrow{\surd v}_e}{c\,e \xrightarrow{\tau}_e \delta(c, v)}$$

Context Rules:

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

$$\frac{e \xrightarrow{\tau}_e e'}{\mathsf{if}\, e \,\mathsf{then}\, f \,\mathsf{else}\, g \xrightarrow{\tau}_e \mathsf{if}\, e' \,\mathsf{then}\, f \,\mathsf{else}\, g}$$

$$\frac{e \xrightarrow{\tau}_e e'}{\mathsf{let}\, x = e \,\mathsf{in}\, f \xrightarrow{\tau}_e \mathsf{let}\, x = e' \,\mathsf{in}\, f}$$

FIGURE 2. Operational Semantics for λ_v

The two relations are defined to be the least ones which satisfy the rules given in Figure 2. The main properties of the operational semantics is captured in the following proposition, whose proof we leave to the reader:

Proposition 2.1 (Subject Reduction). *For every closed expression e in λ_v such that $\vdash e : A$:*

1. *if $e \xrightarrow{\tau}_e e'$ then $\vdash e' : A$*
2. *if $e \xrightarrow{\surd v}_e$ then $\vdash v : A$* □

In addition, we can show that reduction of expressions is deterministic:

Proposition 2.2 (Determinacy). *For every closed expression e in λ_v:*

1. *if $e \xrightarrow{\tau}_e e'$ and $e \xrightarrow{\tau}_e e''$ then $e' = e''$*
2. *if $e \xrightarrow{\surd v}_e$ and $e \xrightarrow{\surd v'}_e$ then $v = v'$* □

2.2 A communicate-by-value process calculus

In the previous section we presented a language λ_v for sequential computation. We now present a language for concurrent computation, where the data commu-

$$\frac{}{\vdash \mathbf{0} : \pi} \qquad \frac{\vdash v : A, \vdash p : \pi}{\vdash k_A!v.\, p : \pi} \qquad \frac{\vdash p : \pi}{\vdash k_B?x_B.\, p : \pi}$$

$$\frac{\vdash p : \pi \quad \vdash q : \pi}{\vdash p + q : \pi} \qquad \frac{\vdash p : \pi \quad \vdash q : \pi}{\vdash p \Vdash q : \pi}$$

FIGURE 3. Type Rules for processes.

nicated on channels are values taken from λ_v. In the next section we will show how these two languages can be combined to give a concurrent λ-calculus.

The syntax of the process language is given by the following grammar:

$$p, q \ldots \in Proc ::= p \Vdash p \mid p + p \mid \mathbf{0} \mid \tau.p \mid k_A?x_A.\, p \mid k_A!v.\, p$$

The process $p \Vdash q$ represents two computation threads running concurrently—in this language p and q are treated symmetrically, but in the next section we introduce the notion of *main thread of computation*, so we use an asymmetric notation for parallel composition.

From *CCS* we adopt process *summation* $+$, the *deadlocked* process, τ prefix, and two constructs for the transmission and reception of values along channels, $k?x.p$ and $k!v.p$; we assume that for each type A, k_A ranges over an infinite set of channels $Chan_A$. The input prefix is a variable binding operator in that in the expression $k?x.\, p$ all free occurrences of x in p are bound.

Processes are typed with judgements $\vdash p : \pi$, given in Figure 3.

We now discuss the operational semantics of processes. We have three possible reductions for a process, based on the labelled transition system for *CCS*:

- $p \xrightarrow{\tau}_p p'$, meaning, as before, that one evaluation step reduces p to p'. However here, $\xrightarrow{\tau}_p$ will model communication between independent evaluation threads. For example we will have $k!v.\, e \Vdash k?x.\, f \xrightarrow{\tau}_p e \Vdash f[v/x]$.
- We have the new relation $p \xrightarrow{k_A!v}_p p'$, meaning that a first possible step in the evaluation of the expression p consists of the emission of a value v, along the channel k_A and the computation can subsequently proceed by evaluating the expression p'. Communication between threads is *synchronous* and so this computation thread can only proceed if there is another concurrent thread which wishes to input a value along this channel.
- The final relation is of the form $p \xrightarrow{k_A?}_p \lambda x_A.\, p'$, meaning that the computation thread corresponding to e can input a value along the channel k_A. This value is of type A and is represented by the free variable x_A in p'. In the terminology of [9] this represents a *late* operational semantics.

In fact the relation $\xrightarrow{k?_A}_p$ will be defined indirectly, in terms of more technically convenient relation $\xrightarrow{k_A?x_A}_p$, defined as $p \xrightarrow{k_A?x_A}_p p'$ iff $p \xrightarrow{k_A?}_p \lambda x_A.\, p'$.

Communication Rules:

$$\frac{}{k!v.\, p \xrightarrow{k!v}_p p} \qquad \frac{}{k?x.\, e \xrightarrow{k?x}_p e}$$

$$\frac{p \xrightarrow{k!v}_p p' \quad q \xrightarrow{k?x}_p q'}{p \Vdash q \xrightarrow{\tau}_p p' \Vdash q'[v/x]} \qquad \frac{p \xrightarrow{k?x}_p p' \quad q \xrightarrow{k!v}_p q'}{p \Vdash q \xrightarrow{\tau}_p p'[v/x] \Vdash q'}$$

Dynamic rules:

$$\frac{p \xrightarrow{\mu}_p p'}{p + q \xrightarrow{\mu}_p p'} \qquad \frac{q \xrightarrow{\mu}_p q'}{p + q \xrightarrow{\mu}_p q'}$$

Context Rules:

$$\frac{p \xrightarrow{\mu}_p p'}{p \Vdash q \xrightarrow{\mu}_p p' \Vdash q} \qquad \frac{p \xrightarrow{\mu}_p p'}{p \Vdash q \xrightarrow{\mu}_p p \Vdash q'}$$

FIGURE 4. Operational Semantics for processes

These relations are defined to be the least relations over closed processes which satisfy the rules given in Figure 4. In these rules we use:

$$a ::= k_A!v \mid k_A?x_A \qquad \mu ::= a \mid \tau$$

We have the following Subject Reduction property:

Proposition 2.3 (Subject Reduction). *For every process p such that $\vdash p : \pi$*

1. *$p \xrightarrow{\tau}_p p'$ implies $\vdash p' : \pi$*
2. *$p \xrightarrow{k_B!v}_p p'$ implies $\vdash v : B$ and $\vdash p' : \pi$*
3. *$p \xrightarrow{k_B?x_B}_p p'$ implies $\vdash p' : \pi$*

Proof By rule induction on the relations involved. □

2.3 Merging the λ and process calculi

In this section we unify the two languages considered in the previous sections. There are numerous ways in which one can conceive of such an unification. For example, as has been pointed out in [1], the language *Facile*, [3], may be considered as a call-by-value λ-calculus, such as λ_v, extended with an extra type process. The syntax of expressions is then extended by various primitives for expressions of this new type, such as a parallel operator $||$, operators for input and output on communication channels, and λ_v is used to provide values which are transmitted and received by objects of this type. Thus in *Facile*, in particular in the abstract version studied in [1], there is a clear distinction between

processes, expressions of type process, and the expressions in the underlying λ-calculus. For example the parallel operator || can only be applied to processes, i.e. expressions of type process.

By contrast, in *CML* [10] every expression is considered to be a *thread of computation*, and expressions can spawn concurrent threads. At any one time there is a main thread of computation, whose result will be returned if that thread terminates. We represent this using the asynchronous parallel operator $p \Vdash q$, which specifies that q is the main thread of computation. For example $\mathsf{true} \Vdash 1$ will return the result 1 and discard the result true. This is reflected in the typing of our extended language; the type of $e \Vdash f$ is given by that of f.

In *CML* there is still a distinction between processes (which can be placed in $+$ contexts) and expressions (which cannot). This is given by the event type constructor. In this paper for simplicity we will use a separate syntactic category for processes rather than a separate type—the full story is given in [2].

We extend the language of expressions by including all processes, and parallel composition of expressions:

$$\begin{array}{rll} e ::= & v \mid c\,e \mid \mathsf{if}\,e\,\mathsf{then}\,e\,\mathsf{else}\,e \mid \mathsf{let}\,x_A{=}e\,\mathsf{in}\,e \mid e\,e & \text{(as before)} \\ \mid & p \mid e \Vdash e & \text{(new)} \end{array}$$

We extend the language of processes by allowing any expression to be used as a process *as long as it is prefixed*:

$$\begin{array}{rll} p ::= & p \Vdash p \mid p + p \mid \mathbf{0} & \text{(as before)} \\ \mid & \tau.e \mid k_A?x_A.e \mid k_A!v.e & \text{(new)} \end{array}$$

For example true is not a valid process, but $\tau.\mathsf{true}$ is. The restriction on the use of expressions in processes is to ensure that weak bisimulation will be a congruence for expressions.

Note that output is restricted to being of values $k_A!v.e$ rather than expressions $k_A!e.f$. We can define an expression for arbitrary output as syntactic sugar:

$$k!e.f = \mathsf{let}\,x{=}e\,\mathsf{in}\,k!x.f$$

but note that this is an output *expression* rather than an output *process*, so cannot be placed in $+$ contexts.

The typing judgements for the new constructs are given in Figure 5.

We use λ_v^{con} to denote the set of well-typed closed expressions in the new syntax and as before use $(\lambda_v^{con})^\circ$ to denote the open expressions.

The operational semantics for λ_v^{con} is given by unifying the previous operational semantics, and using a labelled transition system with labels:

$$a ::= k_A!v \mid k_A?x_A \qquad \mu ::= a \mid \tau \qquad l ::= \mu \mid \surd v$$

In particular, expressions can spawn subprocesses before returning a value, so

$$\frac{}{\vdash \mathbf{0} : A} \qquad \frac{\vdash v : B, \vdash e : A}{\vdash k_B!v.e : A} \qquad \frac{\vdash e : A}{\vdash k_B?x_B.e : A}$$

$$\frac{\vdash p : A \quad \vdash q : A}{\vdash p + q : A} \qquad \frac{\vdash e : A \quad \vdash f : B}{\vdash e \Vdash f : B}$$

FIGURE 5. Type Rules for the merged language

$\xrightarrow{\surd v}$ transitions now have to have a residual, for example:

$$p \Vdash \mathsf{true} \xrightarrow{\surd \mathsf{true}} p \Vdash \mathbf{0}$$

These residuals have to be accommodated in other reductions, for example:

$$\mathsf{if}\,p \Vdash \mathsf{true}\,\mathsf{then}\,e\,\mathsf{else}\,f \xrightarrow{\tau} p \Vdash \mathbf{0} \Vdash e$$

We have the following Subject Reduction Theorem:

Theorem 2.4 (Subject Reduction). *For every e in λ_v^{con} such that $\vdash e : A$*

1. *$e \xrightarrow{\tau} f$ implies $\vdash f : A$*
2. *$e \xrightarrow{\surd v} f$ implies $\vdash v : A$ and $\vdash f : A$*
3. *$e \xrightarrow{k_B!v} f$ implies $\vdash v : B$ and $\vdash f : A$*
4. *$e \xrightarrow{k_B?} f$ implies $\vdash f : B \to A$*

Proof By rule induction on the relations involved. □

We end this section by examining the properties of value production, the relation $\xrightarrow{\surd v}$.

Proposition 2.5. *The operational semantics of λ_v^{con} satisfies the following properties:*

- *single-valuedness: If $e \xrightarrow{\surd v} e'$ then $e' \xrightarrow{\surd w}$ for no w.*
- *value-determinacy: $e \xrightarrow{\surd v} e'$ and $e \xrightarrow{\surd w} e''$ implies $e' = e''$ and $v = w$*
- *forward commutativity:*

$$\begin{array}{ccc} e & \xrightarrow{\mu} & e_1 \\ {\scriptstyle \surd v}\downarrow & & \\ e_2 & & \end{array} \quad \textit{implies} \quad \begin{array}{ccc} e & \xrightarrow{\mu} & e_1 \\ {\scriptstyle \surd v}\downarrow & & \downarrow{\scriptstyle \surd v} \\ e_2 & \xrightarrow[\mu]{} & e_3 \end{array}$$

Values:

$$\frac{}{v \xrightarrow{\surd_v} \mathbf{0}}$$

Communication Rules:

$$\frac{}{k!v.e \xrightarrow{k!v} e} \qquad \frac{}{k?x.e \xrightarrow{k?x} e}$$

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

Dynamic rules:

$$\frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p+q \xrightarrow{\mu} q'}$$

Reductions:

$$\frac{e \xrightarrow{\surd \mu x.(\lambda y.g)} e'}{e\,f \xrightarrow{\tau} e' \Vdash \mathsf{let}\, y{=}f\, \mathsf{in}\, g[\mu x.(\lambda y.g)/x]} \qquad \frac{e \xrightarrow{\surd_v} e'}{\mathsf{let}\, x{=}e\, \mathsf{in}\, f \xrightarrow{\tau} e' \Vdash f[v/x]}$$

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

$$\frac{e \xrightarrow{\surd_v} e'}{c\,e \xrightarrow{\tau} e' \Vdash \delta(c,v)}$$

Context Rules:

$$\frac{e \xrightarrow{\mu} e'}{c\,e \xrightarrow{\mu} c\,e'} \qquad \frac{e \xrightarrow{\mu} e'}{\mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g \xrightarrow{\mu} \mathsf{if}\, e'\, \mathsf{then}\, f\, \mathsf{else}\, g}$$

$$\frac{e \xrightarrow{\mu} e'}{\mathsf{let}\, x{=}e\, \mathsf{in}\, f \xrightarrow{\mu} \mathsf{let}\, x{=}e'\, \mathsf{in}\, f} \qquad \frac{e \xrightarrow{\mu} e'}{e\,f \xrightarrow{\mu} e'\,f}$$

$$\frac{e \xrightarrow{\mu} e'}{e \Vdash f \xrightarrow{\mu} e' \Vdash f} \qquad \frac{f \xrightarrow{\mu} f'}{e \Vdash f \xrightarrow{\mu} e \Vdash f'}$$

$$\frac{f \xrightarrow{\surd_v} f'}{e \Vdash f \xrightarrow{\surd_v} e \Vdash f'}$$

FIGURE 6. Operational Semantics for λ_v^{con}

- *backward commutativity:*

$$e \xrightarrow{\surd_v} e_1 \quad \Big\downarrow \mu \quad \textit{implies} \quad \begin{array}{ccc} e & \xrightarrow{\surd_v} & e_1 \\ \mu\downarrow & & \downarrow\mu \\ e_3 & \xrightarrow[\surd_v]{} & e_2 \end{array}$$

Proof Routine induction on the syntax. □

These special properties of $\xrightarrow{\surd_v}$ imply that in some sense the production of values is *asynchronous*; we will later show, using these properties, that if $e \xrightarrow{\surd_v} e'$ then e is semantically equivalent to the expression $e' \Vdash v$. This latter term can produce the value v but no subsequent behaviour can depend on its production.

3 A Semantic Theory

In this section we develop a semantic equivalence for λ_v^{con} based on *bisimulations*. A typical semantic equivalence, $\sim$, is defined by abstracting in some manner from the operational semantics and is usually a relation over closed expressions, in our case over λ_v^{con}. This is extended in the standard way to a relation $\sim^\circ$ over open terms by letting $e \sim^\circ e'$ if $e\rho \sim e'\rho$ for all closed *substitutions* ρ, i.e. type-respecting functions from variables to closed values.

In this section, we will define two semantic equivalences $\approx^h$ and $=^h$ such that:

- $\approx^h$ is a congruence for λ_v^{con} expressions, and $=^h$ is a congruence for λ_v^{con} processes,
- $\approx^h$ generalises the standard theory of equality of the call-by-value λ-calculus,
- $=^h$ generalises Milner's observational congruence, [7], originally developed for *CCS*.

In the first subsection we explain the definition of *higher-order bisimulation equivalence* while some of its properties are investigated in the remainder.

3.1 Higher Order Bisimulations

Recall from [7] that that a relation $\mathcal{R}$ (over closed expressions from λ_v^{con}) is is a *(strong) simulation* if the following diagram, representing a *transfer property*, can be completed:

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

A semantic equivalence between expressions $\sim$ can then be defined as the largest symmetric simulation. There are many reasons why $\sim$ is inadequate as the basis of a semantic theory for λ_v^{con}; some of these are quite general while others are due to the nature of the language λ_v^{con}.

First in order to take into account the fact that $\xrightarrow{\tau}$ actions are unobservable

we will require that the move $e_1 \xrightarrow{l} e_1'$ be matched by a *weak action* $e_2 \stackrel{\hat{l}}{\Longrightarrow} e_2'$, where

- $\stackrel{\varepsilon}{\Longrightarrow}$ is the reflexive transitive closure of $\xrightarrow{\tau}$
- $\stackrel{l}{\Longrightarrow}$ is $\stackrel{\varepsilon}{\Longrightarrow}\xrightarrow{l}$
- $\stackrel{\hat{l}}{\Longrightarrow}$ is $\stackrel{\varepsilon}{\Longrightarrow}$ if $l = \tau$ and $\stackrel{l}{\Longrightarrow}$ otherwise.

In order to ensure that only closed expressions of the same type are related we consider *typed-indexed* relations $\mathcal{R}$, i.e. families of relations $\mathcal{R}_A$ indexed by types A.

The requirement that an l-action be matched by one with exactly the same label is too strong. For example, the expressions $k_A!(\lambda x.1).\mathbf{0}$ and $k_A!(\lambda x.\,\mathsf{succ}0).\mathbf{0}$ are differentiated although it would be difficult to conceive of a context which can distinguish them. The appropriate definition of simulation should compare not only expressions but also labels. To this end, for any type-indexed relation $\mathcal{R}$, define its *extension to labels* $\mathcal{R}^l$ by:

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

We only require labels to be matched up to $\mathcal{R}^{\,l}$ rather than up to syntactic identity.

Unfortunately, the resulting equivalence now identifies all terms in normal form, since all a normal form can do is tick with its own value. We add the extra requirement that $\mathcal{R}$ be *structure preserving*, i.e.:

1. if $v_1\ \mathcal{R}_{A\to B}\ v_2$ then for all closed values $\vdash w : A$ we have $v_1\,w\ \mathcal{R}_B\ v_2\,w$
2. if $v_1\ \mathcal{R}_A\ v_2$ where A is a base type then $v_1 = v_2$.

Definition 3.1. (Higher-Order Weak Simulation) A type-indexed relation $\mathcal{R}$ over extended λ_v^{con} is a *higher-order weak simulation* if it is structure-preserving and the following diagram can be completed:

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

□

Let $\approx^h$ be the largest symmetric higher-order weak simulation. Since the identity relation I is a higher-order simulation and $\mathcal{R}\ \mathcal{R}$ is whenever $\mathcal{R}$ is, it follows that $\approx^h$ is an equivalence relation.

However, as is usual for weak bisimulations and *CCS*, the choice construct $+$ is not preserved by $\approx^h$. For example $\tau.\mathbf{0} \approx^h \mathbf{0}$ but $k_A!1.\mathbf{0} + \tau.\mathbf{0} \not\approx^h k_A!1.\mathbf{0} + \mathbf{0}$.

Fortunately we can adapt the usual remedy of Milner's observational equivalence, [7], to λ_v^{con}.

Definition 3.2. (Higher-Order Observational Equivalence) Let $=^h$ be the smallest symmetric relation such that the following diagram can be completed:

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

□

Theorem 3.3. *$=^h$ is a congruence for λ_v^{con} processes, and $\approx^h$ is a congruence for λ_v^{con} expressions.*

Proof It is easy to establish that it is an equivalence relation and that it is preserved by operators such as $+$ and the various forms of action prefixing. However it is more difficult to prove that it, or indeed $\approx^h$, is preserved by the parallel construct, i.e. $e_i \approx^h f_i$ implies $e_1 \between f_1 \approx^h e_2 \between f_2$. For example if e_i is $k_A?x.g_i$ where $g_1 \approx^h g_2$ and f_i are of the form $k_A!v_i.\mathbf{0}$ where $v_1 \approx^h v_2$ then we need to establish $g_1[v_1/x] \approx^h g_2[v_2/x]$.

The proof uses Howe's technique, [6], and is relegated to the Appendix. □

Theorem 3.4. *$=^h$ is the largest congruence for λ_v^{con} processes contained in $\approx^h$.*

Proof Let $\approx$ be any congruence on λ_v^{con} processes contained in $\approx^h$. To show $\approx$ is contained in $=^h$ it is sufficient to prove that if $p \approx q$ and $p \xrightarrow{\tau} p'$ then $q \stackrel{\tau}{\Longrightarrow} q'$ for some q' such that $p' \approx^h q'$. Since $\approx$ is a congruence, we have $p + k!0 \approx q + k!0$ (for fresh k) and since $p + k!0 \xrightarrow{\tau} p'$ we have $q + k!0 \Longrightarrow q'$ and $p' \approx^h q'$. Since p' cannot perform k, neither can q', so we must have $q \stackrel{\tau}{\Longrightarrow} q'$. The result follows.

□

It follows immediately that

Corollary 3.5. *In λ_v^{con}:*

- *$e \approx^h f$ implies $\mu.e =^h \mu.f$*
- *$e \approx^h f$ implies $e =^h f$ or $\tau.e =^h f$ or $e =^h \tau.f$.* □

3.2 Properties of λ_v^{con} expressions

We first examine λ_v^{con} as a call-by-value λ-calculus, by considering λ_v^{con} expressions up to weak bisimulation.

It is straightforward to show

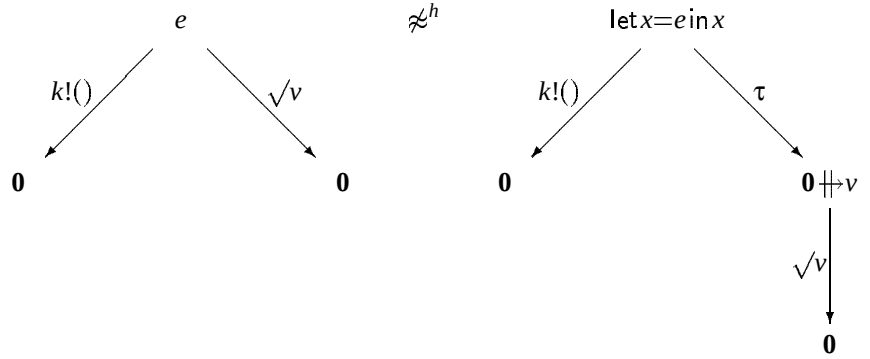
$$(\lambda x.e)\,v \approx^h e[v/x]$$

$$(\mu x.(\lambda y.e))\,v \approx^h e[\mu x.(\lambda y.e)/x][v/y]$$
$$\mathsf{let}\,x{=}v\,\mathsf{in}\,e \approx^h e[v/x]$$
$$\mathsf{let}\,y{=}(\mathsf{let}\,x{=}e\,\mathsf{in}\,f)\,\mathsf{in}\,g \approx^h \mathsf{let}\,x{=}e\,\mathsf{in}\,(\mathsf{let}\,y{=}f\,\mathsf{in}\,g) \quad \text{where } x \notin fv(g)$$

The last two are the left unit and associativity axioms of the monadic meta-language of [8]. The third unit equation:

$$\mathsf{let}\,x{=}e\,\mathsf{in}\,x = e$$

is more difficult to establish. Indeed as pointed out in [2] this is not true in arbitrary labelled transition systems, as can be seen from the following example:



However in the labelled transition system generated by extended λ_v^{con} we have:

Proposition 3.6. *If* $e \xrightarrow{\sqrt{v}} e'$ *then* $e =^h e' \Vdash v$.

Proof The following is a higher-order bisimulation:

$$\{(e, e' \Vdash v) \mid e \xrightarrow{\sqrt{v}} e'\} \cup \{(e', e' \Vdash \mathbf{0}) \mid e \xrightarrow{\sqrt{v}} e'\}$$

Establishing this requires Proposition 2.5. □

Corollary 3.7. $e \approx^h \mathsf{let}\,x{=}e\,\mathsf{in}\,x$

Proof Using the previous Proposition one can show that

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

is a higher-order bisimulation. □

These identities all involve the equivalence $\approx^h$ but using the first part of Corollary 3.5 they can be turned into identities for $=^h$. So for example we have

$$\tau.e =^h \tau.\mathsf{let}\,x{=}e\,\mathsf{in}\,x.$$

The second part of this Corollary indicates that by analysing the initial τ actions we can sometimes come up with slightly stronger identities. Examples of these

are:

$$(\lambda x.e)\,v =^h \tau.e[v/x]$$
$$\mathsf{let}\,x{=}v\,\mathsf{in}\,e =^h \tau.e[v/x]$$

We leave the reader to transform the other identities for $\approx^h$ given above to identities for $=^h$.

3.3 Properties of λ_v^{con} processes

We now turn our attention to the language viewed as a process algebra. The essential features of a process algebra such as *CCS* are a choice operator, a parallel operator and a notion of *action prefixing*. All of these appear in λ_v^{con}. We can take the syntactic expressions of the form $k_A!v.\,, k_A?x.\,, \tau.$ to be action prefixes, ranged over by μ. Note that even if $\mu.e$ is a closed expression e may be open. The following two τ-law of CCS, [7], are valid:

$$p + \tau.p =^h \tau.p$$
$$\mu.e =^h \mu.\tau.e$$

The third τ-law

$$\mu.(p + \tau.q) = \mu.q + \mu.(p + \tau.q)$$

is not in general true; but this is as expected as we have used a *late* operational semantics, [9], and this law does not hold in value-passing *CCS* for such a semantics. It is only satisfied for the τ prefix, and in this case it is derivable from the first law.

The choice operator satisfies the expected laws, those of a commutative monoid:

$$\mathbf{0} + p =^h p$$
$$p + p =^h p$$
$$(p_1 + p_2) + p_3 =^h p_1 + (p_2 + p_3)$$
$$p_1 + p_2 =^h p_2 + p_1$$

The parallel operator does not quite satisfy all the laws of *CCS*. It does satisfy:

$$\mathbf{0} \Vdash e =^h e$$
$$(e_1 \Vdash e_2) \Vdash e_3 =^h e_1 \Vdash (e_2 \Vdash e_3)$$
$$(e_1 \Vdash e_2) \Vdash e_3 =^h (e_2 \Vdash e_1) \Vdash e_3$$

It is not in general symmetric because of its interaction with the production of values; for example $1 \Vdash \mathbf{0} =^h \mathbf{0}$ but $\mathbf{0} \Vdash 1 =^h 1$.

In *CCS* every closed expression is semantically equivalent to a *sum-form*, i.e.

an expression of the form

$$\sum_{i\in I}\mu_i.e_i$$

In λ_v^{con} we can also show that every expression (resp. process) is equivalent, up to $\approx^h$ (resp. $=^h$), to a such a form. This is the subject of the next subsection.

3.4 A head normal form for closed finite expressions

Here we outline a characterisation of $=^h$ in terms of equations and proof rules. The characterisation is restricted to *closed finite expressions* from the language λ_v^{con}, i.e. closed expressions in which for all occurrences of $\mu x.(\lambda y.e)\ x$ does not occur free in e; as already explained in Section 2 such expressions will be denoted by $\lambda y.e$.

The characterisation can be viewed as an extension of the equational characterisation of finite *CCS* expressions, [7]. These equations are given in Figure 7 although the third τ-law from [7] is missing because we are dealing with a *late* behavioural equivalence, together with the usual structural rules for equational reasoning.

The syntax of λ_v^{con} is larger than that of *CCS* and we need extra equations for each of the new syntactic constructs. These are given in Figure 8 and are of five kinds. The first gives the properties of parallelism as an associative operator with left units. The second gives the β-reduction rules for each of the operators. The others show how the process operators (parallelism, summation and prefixing) distribute through the functional operators (if, let and application).

Let $\vdash e = f$ denote that $e = f$ can be deduced in the resulting proof system. We leave the reader to check that this is sound, i.e. $\vdash e = f$ implies $e =^h f$. We will show that any closed finite expression can be converted into *head normal form*, that is:

$$\sum_i \mu_i.e_i \quad \text{or} \quad (\sum_i \mu_i.e_i \between)v$$

Note that we *cannot* use this directly to show completeness of the proof system, since the terms e_i may be open, and we can only normalize closed terms. A complete proof system would require techniques taken from *symbolic bisimulation* [5], which is left for future work.

Proposition 3.8. *For every closed finite expression e there is a head normal form f such that* $\vdash e = f$.

Proof (Outline) Show by induction on syntax that any process can be converted to the form $\sum_i \mu_i.e_i$ and that any expression can be converted to the form $(\sum_i \mu_i.e_i)\ [\between v]$, where $[\between v]$ denotes an optional occurrence of $\between v$. The equational properties of $\between$ given by the 'parallelism' and 'process spawning' rules are needed only in the case of expressions whose head normal form is of the form $p \between v$. □

Summation:

$$\begin{aligned} p+\mathbf{0} &= p \\ p+p &= p \\ p+q &= q+p \\ p+(q+r) &= (p+q)+r \end{aligned}$$

$$\begin{aligned} p+\tau.p &= \tau.p \\ \mu.e &= \mu.\tau.e \end{aligned}$$

Interleaving law:

Let p, q denote $\sum_i \mu_i.e_i, \sum_j \nu_j.f_j$, where $fv(\mu_i) \cap fv(\nu_j) = \emptyset$.

$$\begin{aligned} p \between q = &\sum_i \mu_i.(e_i \between f) + \sum_j \nu_j.(e \between f_j) \\ &+ \sum_{\mu_i = k!v, \nu_j = k?x} \tau.(e_i \between f_j[v/x]) \\ &+ \sum_{\mu_i = k?x, \nu_j = k!v} \tau.(e_i[v/x] \between f_j) \end{aligned}$$

FIGURE 7. Equations for *CCS*

4 Conclusions

In this paper we have described one method for combining the concurrency features of *CCS* with those of the typed λ-calculus. The resulting semantic theory can be viewed as an extension of the theory of bisimulation congruence of *CCS*, when restricted to the syntactic class of *processes*, and when applied to *expressions* as an extension of the theory of equality for call-by-value λ-calculus.

In section 3.4 we have briefly investigated a syntactic characterisation of this semantic theory using a combination of process algebra equations and those from the theory of equality for call-by-value λ-calculus. It is not surprising that these equations are incomplete even for expressions without recursion as the values being communicated may be higher-order abstractions. However even the intro-

Concurrency:

$$\begin{aligned}
\mathbf{0} \between e &= e \\
v \between e &= e \\
(e \between f) \between g &= e \between (f \between g) \\
(e \between f) \between g &= (g \between e) \between g
\end{aligned}$$

Reduction:

$$\begin{aligned}
\mathsf{if\,true\,then}\ e\ \mathsf{else}\ f &= \tau.e \\
\mathsf{if\,false\,then}\ e\ \mathsf{else}\ f &= \tau.f \\
\mathsf{let}\ x{=}v\ \mathsf{in}\ e &= \tau.e[v/x] \\
(\lambda x.e)\,v &= \tau.e[v/x] \\
c\,v &= \tau.\delta(c,v)
\end{aligned}$$

Process spawning:

$$\begin{aligned}
\mathsf{if}\,(e \between e')\,\mathsf{then}\ f\ \mathsf{else}\ g &= e \between (\mathsf{if}\ e'\ \mathsf{then}\ f\ \mathsf{else}\ g) \\
\mathsf{let}\ x{=}(e \between e')\,\mathsf{in}\ f &= e \between (\mathsf{let}\ x{=}e'\,\mathsf{in}\ f) \\
(e \between e')\,f &= e \between (e'\,f) \\
c\,(e \between e') &= e \between (c\,e')
\end{aligned}$$

Choice:

$$\begin{aligned}
\mathsf{if}\,(p+q)\,\mathsf{then}\ f\ \mathsf{else}\ g &= (\mathsf{if}\ p\,\mathsf{then}\ f\ \mathsf{else}\ g) + (\mathsf{if}\ q\,\mathsf{then}\ f\ \mathsf{else}\ g) \\
\mathsf{let}\ x{=}(p+q)\,\mathsf{in}\ f &= (\mathsf{let}\ x{=}p\,\mathsf{in}\ f) + (\mathsf{let}\ x{=}q\,\mathsf{in}\ f) \\
(p+q)\,f &= (p\,f) + (q\,f) \\
c\,(p+q) &= (c\,p) + (c\,q)
\end{aligned}$$

Prefixing:

$$\begin{aligned}
\mathsf{if}\ \mu.e\,\mathsf{then}\ f\ \mathsf{else}\ g &= \mu.(\mathsf{if}\ e\,\mathsf{then}\ f\ \mathsf{else}\ g) \\
\mathsf{let}\ x{=}\mu.e\,\mathsf{in}\ f &= \mu.(\mathsf{let}\ x{=}e\,\mathsf{in}\ f) \\
(\mu.e)\,f &= \mu.(e\,f) \\
c\,(\mu.e) &= \mu.(c\,e)
\end{aligned}$$

FIGURE 8. Extra equations

duction of rules such as

$$\frac{\text{for all } v,\ (\lambda x.e_1)v = (\lambda y.e_2)v}{\lambda x.e_1 = \lambda y.e_2} \qquad \frac{\text{for all } v,\ (\lambda x.e_1)v = (\lambda y.e_2)v}{k_A?x.e_1 = k_A?y.e_2}$$

will not lead to a complete characterisation. The problem is that the behaviour of a process such as $k_A?x.e$ of type B depends on the behaviour of $(\lambda x.e)v$ where v ranges over all values of type B. However B may be any type, and in particular a type which has A as a sub-type, and therefore the proof rules above are not of great help in establishing judgements of the form $k_A?x.e_1 = k_A?y.e_2$. In order to obtain a complete syntactic characterisation the proof system needs to be generalised to *open expressions*, where the symbolic techniques of [5] may be applicable.

We have concentrated on a *communicate-by-value* paradigm because this is the approach taken in languages such as *CML* and *Facile*. However in *HO*π, [11], and *CHOCS*, [12], actual text or code is transmitted between processes, much in the same way as with standard β-reduction in the λ-calculus. In *CML* there is also a mechanism, using Event types, for this kind of data exchange and in *Facile* scripts are used in a similar manner. To study this form of communication an alternative to λ_v^{con}, say λ_n^{con}, could be designed with reductions such as:

$$(\lambda x.e_1)e \xrightarrow{\tau} e_1[e/x], \qquad k_A!e.e_1 \between k_A?x.e_2 \xrightarrow{\tau} e_1 \between e_2[e/x]$$

We conjecture that in this language $\approx^h$ and $=^h$ are still congruences for *expressions* and *processes* respectively. Nevertheless it is known from [12] that higher-order bisimulation congruence is not preserved by all *CHOCS* contexts, the reason being that in *CHOCS* no distinction is made between *processes* and *expressions*. In λ_v^{con}, and the hypothetical λ_n^{con}, variables are *expressions* and may only be instantiated by expressions. If however λ_n^{con} were further extended to allow variables to be instantiated by *processes*, and therefore the communication of *processes*, then $\approx^h$, $=^h$ would no longer be preserved by all contexts. As a counterexample consider the two *processes* p_1, p_2 defined by $k_A!\mathbf{0}.\mathbf{0}$, $k_A!(\tau.\mathbf{0}).\mathbf{0}$ respectively. These two *processes* send the *processes* $\mathbf{0}$, $\tau.\mathbf{0}$, respectively, along the channel k_A and it is straightforward to verify $p_1 =^h p_2$. However $C[p_1] \neq^h C[p_2]$ where C is the context $k_A?x.(x + l!0.\mathbf{0}) \between C[\,]$.

This phenomenon also occurs in *CML* and is studied in [2]. A suitable modification of higher-order bisimulations equivalence called *hereditary* bisimulations is proposed and shown to be preserved by all *CML* contexts. This seems to be the most appropriate form of bisimulation for λ_n^{con} extended with *process*-communication. However we leave this for further study.

A Appendix: Congruence proofs

We prove Theorem 3.3 using a variant of Howe's [6] technique, and following Gordon's [4] presentation. The proof follows closely that in Section 5 of [2] and here we merely state the required propositions.

One-level deep contexts are defined by:

$$D ::= x \mid l \mid c\cdot_1 \mid \mathsf{if}\cdot_1\mathsf{then}\cdot_2\mathsf{else}\cdot_3 \mid \mathsf{let}x{=}\cdot_1\mathsf{in}\cdot_2 \mid \cdot_1\cdot_2 \mid \mu x.(\lambda y.\cdot_1)$$
$$k_A?x.\cdot_1 \mid k_A!\cdot_1.\cdot_2 \mid \cdot_1+\cdot_2 \mid \cdot_1\sharp\cdot_2 \mid \tau.\cdot_1$$

Let D_n range over *restricted* one-level deep contexts: one-level deep contexts which do not use $+$.

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 by:

$$\begin{aligned}
\widehat{\mathcal{R}}^n &= \{(D_n[\vec{e}], D_n[\vec{f}]) \mid e_i\ \mathcal{R}^n\ f_i\} \cup \widehat{\mathcal{R}}^s \\
\widehat{\mathcal{R}}^s &= \{(D[\vec{e}], D[\vec{f}]) \mid e_i\ \mathcal{R}^s\ f_i\} \\
&\quad \cup \{(\mu x.(\lambda y.e), \mu x.(\lambda y.f)) \mid e\ \mathcal{R}^n\ f\} \\
&\quad \cup \{(\tau.e, \tau.f) \mid e\ \mathcal{R}^n\ f\} \\
&\quad \cup \{(k?x.e, k?x.f) \mid e\ \mathcal{R}^n\ f\} \\
&\quad \cup \{(k!v.e, k!w.f) \mid e\ \mathcal{R}^n\ f, v\ \mathcal{R}^n\ w\}
\end{aligned}$$

The following proposition is easily established, using induction on contexts.

Proposition A.1. *If $\mathcal{R}$ is an equivalence and $\widehat{\mathcal{R}} \subseteq \mathcal{R}$, then $\mathcal{R}^s$ is a congruence on processes and $\mathcal{R}^n$ is a congruence on expressions*

Proof The proof is by structural induction on contexts. The definitions of $\widehat{\mathcal{R}}^n$ and $\widehat{\mathcal{R}}^s$ are intended to reflect the syntactic interplay between expressions and processes; two weakly bisimilar expressions become weakly congruent when guarded, or 'thunked'. For example, if $e\ \mathcal{R}^n\ f$, and $C = k_A?x.\cdot_1 + p$, then $C[e]\ \mathcal{R}^n\ C[f]$ because even though C is not a restricted context, we have $C[e]\ \widehat{\mathcal{R}}^s\ C[f]$ and $\widehat{\mathcal{R}}^s \subseteq \widehat{\mathcal{R}}^s \subseteq \mathcal{R}^n$. □

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

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

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 given in the three following Propositions. Their proofs are simple variants of the corresponding theorems in [4, 6] and are nearly identical to those in Section 5 of [2].

Proposition A.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. $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$. □

Proposition A.3. *If $\mathcal{R}$ is a preorder then for any $v\ \mathcal{R}^{\bullet n}\ w$: if $e\ \mathcal{R}^{\bullet n}\ f$ then $e[v/x]\ \mathcal{R}^{\bullet n}\ f[w/x]$.*

Proof Suppose $v\ \mathcal{R}^{\bullet n}\ w$; since $\widehat{\mathcal{R}}^s \subseteq \widehat{\mathcal{R}}^n$, the proof is a by a mutual induction on the following two statements:

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

These are shown by structural induction on e. □

Proposition A.4. *If $\mathcal{R}$ is an equivalence then $\mathcal{R}^{\bullet *}$ is symmetric.* □

We need one specific property of the compatible closure of bisimulation:

Proposition A.5. *When restricted to closed expressions, $\approx^{h\bullet}$ is a simulation, and if $e =^{h\bullet} f$ and $e \xrightarrow{l_1} e'$ then $f \stackrel{l_2}{\Longrightarrow} f'$ where $l_1 \approx^{h\bullet} l_2$ and $e' \approx^{h\bullet} f'$.*

Proof Very similar to the proof of Proposition 4.4 of [2]. □

We now have all the information we need to prove Theorem 3.3.

Theorem A.6. *$=^h$ is a congruence for λ_v^{con} processes, and $\approx^h$ is a congruence for λ_v^{con} expressions.*

Proof Follows from Propositions A.2, A.5 and A.4. □

References

[1] Roberto Amadio. From a concurrent λ-calculus to the π-calculus. In *FCT'95*. Springer–Verlag, 1995.

[2] W. Ferreira, M. Hennessy, and A. Jeffrey. A Theory of Weak Bisimulation for Core CML. Technical Report 5/95, COGS, University of Sussex, 1995.

[3] A. Giacalone, P. Mishra, and S. Prasad. A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[4] A. Gordon. Bisimilarity as a theory of functional programming. In *Proceediings of MFPS95*, number 1 in Electronic Notes in Computer Science. Springer–Verlag, 1995.

[5] M. Hennessy and H. Lin. A proof system for value passing processes. In *Proc. of CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 202–216, 1993. To appear in *Acta Informatica*.

[6] D. Howe. Equality in lazy computation systems. In *Proceedings of LICS89*, pages 198–203, 1989.

[7] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[8] Eugenio Moggi. Notions of computation and monad. *Information and Computation*, 93:55–92, 1991.

[9] J. Parrow and D. Sangiorgi. Algebraic theories for value-passing calculi. Technical report, University of Edinburgh, 1993. Also Technical Report from SICS, Sweeden.

[10] J. H. Reppy. A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN 91 PLDI*, number 26 in SIGPLAN Notices, pages 294–305, 1991.

[11] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. Phd thesis, Edinburgh University, Scotland, 1992.

[12] B. Thomsen. Higher order communicationg systems theory. *Information and Computation*, 116:38–57, 1995.