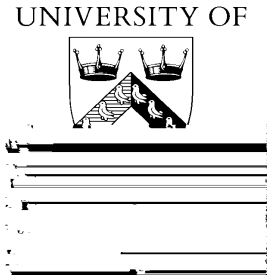
UNIVERSITY OF SUSSEX

COMPUTER SCIENCE



Combining the typed λ-ca cu us with CC

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

Report / May

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

ISSN –

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 significance 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 prefixing $a?, a!$. These action prefixes 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 defined 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 significance. 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:

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

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

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

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

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

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

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

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

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

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:

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

The process $p \between q$ 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 $k?x$

processes, expressions of type process, and the expressions in the underlying λ-calculus. For example the parallel operator ||

Values:

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

Communication Rules:

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

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

Dynamic rules:

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

Reductions:

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

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

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

Context Rules:

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

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

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

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

FIGURE 6. Operational Semantics for λ_v^{con}

- *backward commutativity*

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

Proof Routine induction on the syntax. □

These special properties of $\xrightarrow{\surd v}$ imply that in som053.641(r)2.8340875537.423Tm[(51.01672(o)4.7

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:

$$\begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ & & \end{array} \qquad\qquad \begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ & & \end{array}$$

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

$$\mathsf{let}\, x = v \,\mathsf{in}\, e \approx^h e[v/x]$$

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

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. *A head normal form for closed finite expressions*

Here we outline a characterisation of $=^h$ in terms of equations and proof rules.

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:

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

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:

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

The fon

[6] D. Howe. Equality in lazy computation systems. In *Proceedings of LICS* , 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 1 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.