A Calculus of Higher Order Communicating Systems

Bent Thomsen
Department of Computing
Imperial College of Science and Technology
180 Queen's Gate, London SW7 2BZ, England

Abstract

In this paper we present *A Calculus of Higher Order Communicating Systems.* This calculus considers sending and receiving processes to be as fundamental as nondeterminism and parallel composition.

The calculus is an extension of CCS [Mil80] in the sense that all the constructions of CCS are included or may be derived from more fundamental constructs and most of the mathematical framework of CCS carries over almost unchanged.

Clearly CCS with processes as first class objects is a powerful metalanguage and we show that it is possible to simulate the untyped λ-calculus in CHOCS. The relationship between CHOCS and the untyped λ-calculus is further strengthened by a result showing that the recursion operator is unnecessary in the sense that recursion can be simulated by means of process passing and communication.

As pointed out by R. Milner in [Mil80], CCS has its limitations when one wants to describe unboundedly expanding systems as e.g. an unbounded number of procedure invocations in an imperative concurrent programming language. We show how neatly CHOCS may describe both call by value and call by reference parameter mechanisms for the toy language of chapter 6 in [Mil80].

1 Introduction

A calculus for computation should provide a mathematical framework for the description of and reasoning about computing systems all inside the calculus. To guide the human mind when reasoning about the complex nature of nondeterministic and concurrent systems R. Milner devised *A Calculus of Communicating Systems* as described in [Mil80] and later refined in [Mil83, HenMil85]. This calculus is intended to provide a minimal set of constructions for description of nondeterministic and concurrent systems. According to [Mil80], one of the original intentions of CCS was that it should serve as the λ-calculus of concurrent systems. Subsequent research shows that it serves well as such for a large range of applications. But, as already pointed out in [Mil80], it has its limitations when one wants to describe unboundedly expanding systems as e.g. an unbounded number of procedure invocations in an imperative concurrent programming language.

We believe that this deficiency comes from the first order nature of CCS although a later extension of CCS [Mil83], allowing dynamic use of communication channels to be encoded by subscripts and value passing, may be used to take care of the above problem. This solution has been further refined in [EngNie86], where labels or portnames are considered as first class objects. Both these solutions seem low level and far removed from the λ-calculus analogy, since they do not explicitly support any higher order constructions.

Higher order constructs arise in almost any branch of theoretical computer science, since they yield elegant and powerful abstraction techniques. But so far almost all theories about nondeterministic and concurrent systems have a first order nature. A few attempts to allow higher order constructs in notions for description of nondeterministic and concurrent systems have earlier been put forward such as [KenSle83, AstReg87]. Unfortunately these theories are rather complicated and to some extent too informal and undeveloped, so their applications have been limited. Only recently some more promising treatments of processes as first class objects have been proposed [Bou88, Nie88, KenSle88, Chr88]. In [Bou88] a CCS-like language with special operators and process passing is used to describe the λ-calculus. [Nie88] presents a mixture of a typed λ-calculus and a CCS-like language with processes as first class objects. It is shown how the types (including sorts of processes) of programs may be used to detect certain errors statically. Both [KenSle88] and [Chr88] focus on formulating denotational semantics for CCS respectively CSP with processes as communicable values.

In the following we put forward *A Calculus of Higher Order Communicating Systems* which considers sending and receiving processes to be as fundamental as nondeterminism and parallel composition. The calculus is an extension of CCS in the sense that all the constructions of CCS are included or may be derived from more fundamental constructs and most of the mathematical framework of CCS carries over almost unchanged. The calculus is given an operational semantics by means of a labeled transition system [Plo81]. The fundamental bisimulation equivalence [Par81, Mil83] of CCS — commonly accepted as the finest extensional or behavioural equivalence between processes one would impose — is extended to take processes as communicable values into account. Clearly CCS with processes as first class objects is a powerful metalanguage and we show that it is possible to simulate the untyped λ–calculus in CHOCS. The relationship between CHOCS and the untyped λ-calculus is further strengthened because we do not need an explicit recursion operator to obtain infinite behaviours, since the recursion operator can be simulated by means of process passing and communication.

The outline of this paper is as follows: In section 2 we introduce the well–established notions of labeled transition systems [Plo81] and bisimulation [Par81, Mil83] together with the extensions necessary to take processes as communicable values into account. In section 3 we present a language for description of higher order communicating systems including its operational semantics. Section 4 provides a range of algebraic laws satisfied by the calculus. A main theorem of this section is the simulation of recursion by means of process passing and communication. We briefly comment on the subject of sound and complete proof systems. In section 5 we show how the untyped λ–calculus may be encoded in CHOCS. Section 6 contains an application of CHOCS as a metalanguage defining the semantics of an imperative toy language with parallel constructs first studied in [Mil80]. The main result of this section is the neat handling of procedures, both the call by value and the call by reference parameter mechanisms. Finally section 7 contains some concluding remarks. We comment on the denotational approach to semantics for languages with processes as communicable values and we outline the relationship between the approach of [EngNie86] and CHOCS.

2 Transition Systems and Bisimulation

We take the approach of many recent researchers, especially R. Milner (see e.g. [Mil80, Mil83]), by defining the semantics of concurrent systems by the set of experiments they offer to an observer. I.e. we use the model of *labeled transition systems* [Plo81] as a tool for defining the operational semantics of concurrent systems.

Let Pr be a set of processes and Act a set of actions which processes may perform. A derivation relation $\longrightarrow \subseteq Pr \times Act \times Pr$ defines the dynamic change of processes as they perform actions. For $(p, a, q) \in \longrightarrow$ we normally write $p \xrightarrow{a} q$ which may be interpreted as: "the process p can perform an action a and by doing so become the process q". We use the usual abbreviations as e.g. $p \xrightarrow{a}$ for $\exists q \in Pr.p \xrightarrow{a} q$ and $p \not\xrightarrow{a}$ for $\neg\exists q \in Pr.p \xrightarrow{a} q$. The triple $\mathcal{P} = (Pr, Act, \longrightarrow)$ constitutes the transition system of processes. Based on the operational semantics given by the transition system, several equivalences and preorders have been proposed in order to capture various aspects of the observational behaviour of processes. One of these is the equivalence induced by the notion of bisimulation [Par81, Mil83].

Definition 2.1 *A* bisimulation *R is a binary relation on Pr such that whenever pRq and $a \in Act$ then:*

(i) Whenever $p \xrightarrow{a} p'$, then $q \xrightarrow{a} q'$ for some q' with $p'Rq'$

(ii) Whenever $q \xrightarrow{a} q'$, then $p \xrightarrow{a} p'$ for some p' with $p'Rq'$

Two processes p and q are said to be bisimulation equivalent iff there exists a bisimulation R containing (p, q). In this case we write $p \sim q$.

Now for $R \subseteq Pr^2$ we can define $\mathcal{B}(R)$ as the set of pairs (p, q) satisfying for all $a \in Act$ the clauses (i) and (ii) above. From this definition it follows immediately that R is a bisimulation just in case $R \subseteq \mathcal{B}(R)$. Also, $\mathcal{B}$ is easily seen to be a monotone endofunction on the complete lattice of binary relations (over Pr) under subset inclusion. Standard fixed point results, due to Tarski [Tar55], yield that a maximal fixed point for $\mathcal{B}$ exists and is defined as $\bigcup\{R \mid R \subseteq \mathcal{B}(R)\}$. This maximal fixed point actually equals $\sim$. Moreover, $\sim$ is an equivalence relation and even a congruence with respect to the usual CCS process constructions [Mil80].

So far the set of actions has been uninterpreted. As we shall see later we want to use the model of labeled transition systems to give semantics to a process language with processes as communicable values.

In [Mil80] transitions of the form $p \xrightarrow{av} p'$, where v is some value, are used to give semantics to CCS with value passing in communication. We shall pursue this idea and from now on we shall consider labeled transition systems $\mathcal{P} = (Pr, Act, \longrightarrow)$, where Act has the form $Names \times \{?, !\} \times Pr \cup \{\tau\}$ and $Names$ is an uninterpreted set referred to as a set of portnames. $p \xrightarrow{a?p'} p''$ may be read as "p can receive the process p' at port a and in doing so become the process p''". $p \xrightarrow{a!p'} p''$ may be read as "p can send the process p' via port a and in doing so become the process p''". Note that instead of insisting on an abelian monoid structure on the set $Names$ of portnames as in [Mil83] we simply use the CSP–like notation of ?, ! to indicate the input/output direction of communication. A special symbol τ not in $Names$ will be used to symbolize internal moves of processes and $p \xrightarrow{\tau} p'$ may be interpreted as "the process p can do an internal or silent move and in doing so become the process p'". Throughout this paper we shall use Γ to stand for any action $a?p$, $a!p$ or τ.

To capture the observational behaviour of processes capable of sending and receiving processes we extend the notion of bisimulation. Bisimulation is commonly accepted as the finest extensional or behavioural equivalence between processes that one would impose and the equivalence corresponds to a view where processes are black boxes only distinguishable by their interaction capabilities in different environments. Therefore the extension of bisimulation should not distinguish between equivalent processes even when they are sent or received in communication.

This is captured in the following definition:

Definition 2.2 *A* higher order bisimulation *R is a binary relation on Pr such that whenever pRq and $\Gamma \in Act$ then:*

(i) Whenever $p \xrightarrow{\Gamma} p'$, then $q \xrightarrow{\Gamma'} q'$ for some q', Γ' with $\Gamma\hat{R}\Gamma'$ and $p'Rq'$

(ii) Whenever $q \xrightarrow{\Gamma} q'$, then $p \xrightarrow{\Gamma'} p'$ for some p', Γ' with $\Gamma\hat{R}\Gamma'$ and $p'Rq'$

Where $\hat{R} = \{(\Gamma,\Gamma') \mid (\Gamma = a?p''\ \&\ \Gamma' = a?q''\ \&\ p''Rq'') \vee (\Gamma = a!p''\ \&\ \Gamma' = a!q''\ \&\ p''Rq'') \vee (\Gamma = \Gamma' = \tau)\}$.
Two processes p and q are said to be higher order bisimulation equivalent iff there exists a higher order bisimulation R containing (p,q). In this case we write $p \sim q$.

We may define $\mathcal{HB}(R)$ for $R \subseteq Pr^2$ as the set of pairs (p,q) satisfying clause (i) and (ii) above. It is easy to see that $\mathcal{HB}$ is a monotone endofunction and that there exists a maximal fixed point for $\mathcal{HB}$. This equals $\sim$. If we interpret transitions like $p \xrightarrow{a} p'$ as $p \xrightarrow{a?p''} p'$ for any p'' it is easy to see that any bisimulation equivalent pair of processes is also higher order bisimulation equivalent and we shall drop the higher order prefix from now on. This justifies the ambiguous use of $\sim$ as well. In the above definition $\hat{R}$ takes care of extending R to the processes passed in communication. As we shall see later this has the effect that we do not distinguish between equivalent processes passed in communication.

Proposition 2.3 *$\sim$ is an equivalence*

When τ-actions are interpreted as unobservable internal actions the bisimulation equivalence between processes is too distinctive. To refine the bisimulation equivalence we need the following derived transition relations:

Definition 2.4

$$p \overset{a?p'}{\Longrightarrow} p'' \equiv p \xrightarrow{\tau}{}^{*} \xrightarrow{a?p'} \xrightarrow{\tau}{}^{*} p''$$
$$p \overset{a!p'}{\Longrightarrow} p'' \equiv p \xrightarrow{\tau}{}^{*} \xrightarrow{a!p'} \xrightarrow{\tau}{}^{*} p''$$
$$p \overset{\varepsilon}{\Longrightarrow} p' \equiv p \xrightarrow{\tau}{}^{*} p'$$

where $p \xrightarrow{\Gamma}\xrightarrow{\Gamma'} p''$ means $\exists p'.p \xrightarrow{\Gamma} p' \& p' \xrightarrow{\Gamma'} p''$ and $\xrightarrow{\tau}{}^{*}$ is the reflexive and transitive closure of $\xrightarrow{\tau}$.

Intuitively we may read $p \overset{a?p'}{\Longrightarrow} p''$ as "after a finite number of internal actions p is in a state where it can receive a process p' on a and in doing so end up in a state p'' after a finite number of internal actions".

Weak higher order bisimulation equivalence or observational equivalence may now be defined:

Definition 2.5 *A* weak higher order bisimulation *R is a binary relation on Pr such that whenever pRq and $\Phi \in (Act \setminus \{\tau\}) \cup \{\varepsilon\}$ then:*

(i) Whenever $p \overset{\Phi}{\Longrightarrow} p'$, then $q \overset{\Phi'}{\Longrightarrow} q'$ for some q', Φ' with $\Phi\hat{\hat{R}}\Phi'$ and $p'Rq'$

(ii) Whenever $q \overset{\Phi}{\Longrightarrow} q'$, then $p \overset{\Phi'}{\Longrightarrow} p'$ for some p', Φ' with $\Phi\hat{\hat{R}}\Phi'$ and $p'Rq'$

Where $\hat{\hat{R}} = \{(\Phi,\Phi') \mid (\Phi = a?p''\ \&\ \Phi' = a?q''\ \&\ p''Rq'') \vee (\Phi = a!p''\ \&\ \Phi' = a!q''\ \&\ p''Rq'') \vee (\Phi = \Phi' = \varepsilon)\}$.
Two processes p and q are said to be weak higher order bisimulation equivalent iff there exists a higher order bisimulation R containing (p,q). In this case we write $p \approx q$.

We may define $\mathcal{WHB}(R)$ for $R \subseteq Pr^2$ as the set of pairs (p,q) satisfying clause (i) and (ii) above. It is easy to see that $\mathcal{WHB}$ is a monotone endofunction and that there exists a maximal fixed point for $\mathcal{WHB}$. This equals $\approx$.

Proposition 2.6 *$\approx$ is an equivalence*

Bisimulation equivalence is more distinctive than observational equivalence which is a direct consequence of the following proposition.

Proposition 2.7 *$p \sim p' \Rightarrow p \approx p'$*

3 Syntax and Semantics

In this section we introduce the syntax and semantics of a language for description of higher order communicating systems together with its operational semantics. CHOCS extends CCS as in [Mil80, Mil83, HenMil85] simply by allowing processes to be both sent and received and equally important; to be used when received.

Processes are built from the inactive process *nil*, two types of action prefixing, often referred to as input guarding and output guarding, (nondeterministic) choice, parallel composition, restriction, renaming and variables to be bound by input prefix:

$$p ::= nil \mid a?x.p \mid a!p'.p \mid p + p' \mid p \mid p' \mid p\backslash a \mid p[S] \mid x$$

where $a \in Names$, $S : Names \rightarrow Names$ and $x \in V$ (a set of variables).

To avoid heavy use of brackets we adopt the following precedence of operators: restriction > prefix > parallel composition > choice.

We denote by Pr the set of processes, i.e. the set of expressions built according to the above syntax. Readers familiar with CCS will notice that there is no recursion construct in CHOCS. We shall later see (Theorem 4.10) how recursive behaviours may be simulated using only process passing in communication.

We focus on the process passing and leave out details about other values. Pure synchronization may be obtained by ignoring the processes being sent and received. We shall prefixing: $a?x.p \xrightarrow{a?p'} p[p'/x] \qquad a!p'.p \xrightarrow{a!p'} p$

choice: $\dfrac{p \xrightarrow{\Gamma} p'}{p+q \xrightarrow{\Gamma} p'} \qquad \dfrac{p \xrightarrow{\Gamma} p'}{q+p \xrightarrow{\Gamma} p'}$

parallel: $\dfrac{p \xrightarrow{\Gamma} p'}{p \mid q \xrightarrow{\Gamma} p' \mid q} \qquad \dfrac{p \xrightarrow{\Gamma} p'}{q \mid p \xrightarrow{\Gamma} q \mid p'} \qquad \dfrac{p \xrightarrow{a?p'} p'' \; q \xrightarrow{a!p'} q''}{p \mid q \xrightarrow{\tau} p'' \mid q''} \qquad \dfrac{p \xrightarrow{a!p'} p'' \; q \xrightarrow{a?p'} q''}{p \mid q \xrightarrow{\tau} p'' \mid q''}$

restriction: $\dfrac{p \xrightarrow{a?p'} p''}{p\backslash b \xrightarrow{a?p'} p''\backslash b}\ , a \neq b \qquad \dfrac{p \xrightarrow{a!p'} p''}{p\backslash b \xrightarrow{a!p'} p''\backslash b}\ , a \neq b \qquad \dfrac{p \xrightarrow{\tau} p''}{p\backslash b \xrightarrow{\tau} p''\backslash b}$

renaming: $\dfrac{p \xrightarrow{a?p'} p''}{p[S] \xrightarrow{S(a)?p'} p''[S]} \qquad \dfrac{p \xrightarrow{a!p'} p''}{p[S] \xrightarrow{S(a)!p'} p''[S]} \qquad \dfrac{p \xrightarrow{\tau} p''}{p[S] \xrightarrow{\tau} p''[S]}$

Table 1: Operational semantics for CHOCS

use the sloppy notation $a?.p$ and $a!.p$ as action prefixing for pure synchronization. Other types of values may — with little theoretical overhead — be obtained simply by encoding the values in pure synchronization using the approach of [Mil83] by introducing a family of value indexed guards and generalizing the (nondeterministic) choice operator. We shall indeed use this technique in section 6 and we refer to this section for further discussion.

Input guards are variable binders. This implies a notion of free and bound variables.

Definition 3.1 *We define the set of free variables $FV(p)$ by induction:*

$$\begin{aligned}
FV(nil) &= \emptyset \\
FV(a?x.p) &= FV(p) \setminus \{x\} \\
FV(a!p'.p) &= FV(p) \cup FV(p') \\
FV(p+p') &= FV(p) \cup FV(p') \\
FV(p \mid p') &= FV(p) \cup FV(p') \\
FV(p\backslash a) &= FV(p) \\
FV(p[S]) &= FV(p) \\
FV(x) &= \{x\}
\end{aligned}$$

A variable which is not free i.e. does not belong to $FV(p)$ is said to be bound in p.

The above definition may be rephrased as: x is free in p if x is not contained in any subexpression $a?x.p'$. An expression p is closed if $FV(p) = \emptyset$. Closed expressions are referred to as programs.

To allow processes received in communication to be used we need a way of substituting the received processes for bound variables.

Definition 3.2 *The substitution $p[q/x]$ is defined structurally on p:*

$$\begin{aligned}
nil[q/x] &\equiv nil \\
(a?y.p)[q/x] &\equiv \begin{cases} a?y.(p[q/x]) & \text{if } y \neq x \text{ and } y \notin FV(q) \\ a?z.((p[z/y])[q/x]) & \text{for some } z \neq y \text{ and } z \neq x \text{ and not free in } q \text{ nor } p \text{ otherwise} \end{cases} \\
(a!p'.p)[q/x] &\equiv a!(p'[q/x]).p[q/x] \\
(p+p')[q/x] &\equiv (p[q/x]) + (p'[q/x]) \\
(p \mid p')[q/x] &\equiv (p[q/x]) \mid (p'[q/x]) \\
(p\backslash a)[q/x] &\equiv (p[q/x])\backslash a \\
(p[S])[q/x] &\equiv (p[q/x])[S] \\
(y)[q/x] &\equiv \begin{cases} q & \text{if } x = y \\ y & \text{otherwise} \end{cases}
\end{aligned}$$

This definition extends the definition of substitution given in [Mil81] by allowing substitution in processes built using the parallel composition, restriction or renaming operators. To a certain extent this definition resembles the definition of substitution in the λ-calculus as defined in [Bar85]. We shall pursue this further in a later section (see lemma 5.3). Note how substitution is straightforward only taking care of change of bound variables, this is in contrast to the very complicated definition of extended substitution given in [EngNie86], where also change of ports restricted by $\backslash a$ is taken into account.

The operational semantics of CHOCS is given in terms of a labeled transition system.

Definition 3.3 *Let $\rightarrow$ be the smallest subset of $Pr \times Act \times Pr$, where $Act = Names \times \{?,!\} \times Pr \cup \{\tau\}$, closed under the rules of table 1.*

A process guarded by input prefix has the capability of receiving any process. The received process is put into use by substituting it for the bound variable. Readers familiar with [Mil80] will recognize the similarities with the operational semantics for input guarding in CCS with value passing. Parallel composition acts either asynchronously interleaved or by synchronized message passing e.g. $a?x.p \mid a!p'.p''$ can perform $a?q$ or $a!p'$ as well as τ. The restriction and renaming operators have no effect on the processes sent or received. This makes sense if we relate processes to running programs in a multiprocessor system since here processes sent as values are just programs sent to another processor. When the received program runs as a process, this process does not necessarily have to have the same restrictions on its communication capabilities as the sending process. Of course we could specify so as in e.g. $(a!(p'\backslash b).p)\backslash b$. This is a matter of choice and other possibilities exist as e.g. in the ECCS-language of [EngNie86] where $\backslash a$ is treated as a kind of binding operator. We do not want to do so since this seems to complicate the theory unduly. One of the motivations for doing so in ECCS is to be able to give a static account of the ports in use extracted from the program text. But as the operational semantics of ECCS shows it is necessary to introduce new portnames in the semantics which reflects that it is not possible to give a static account anyway. Also the connection between input prefix and restriction as binders of portnames in communication is not at all clear. We comment further on the relationship between ECCS and CHOCS in the concluding section.

We may now relate the process constructions of CHOCS to the underlying semantic equivalence $\sim$. The following properties may be verified by straightforward constructions of bisimulations.

Proposition 3.4 *$\sim$ is a congruence relation. That is:*

1. $a?x.p \sim a?x.q$ *if* $p[p'/x] \sim q[q'/x]$ *for all* $p' \sim q'$.
2. $a!p'.p \sim a!q'.q$ *if* $p \sim q$ *and* $p' \sim q'$.
3. $p + p' \sim q + q'$ *if* $p \sim q$ *and* $p' \sim q'$.
4. $p \mid p' \sim q \mid q'$ *if* $p \sim q$ *and* $p' \sim q'$.
5. $p\backslash a \sim q\backslash a$ *if* $p \sim q$.
6. $p[S] \sim q[S]$ *if* $p \sim q$.

This property ensures that a compositional verification strategy exists; If correctness of an implementation *IMP* with respect to a specification *SPEC* is taken to be that of their equivalence i.e. *IMP* $\sim$ *SPEC* we may decompose the task of verification by decomposing the *IMP* into subimplementations $IMP_1 \dots IMP_n$ combined by some operator O of the language (i.e. $IMP = O[IMP_1 \dots IMP_n]$). We then have to find subspecifications $SPEC_i$ such that $IMP_i \sim SPEC_i$ and show $O[SPEC_1 \dots SPEC_n] \sim SPEC$. Then the result follows from substitutivity of $\sim$. This strategy may be applied repeatedly until the task of verification is broken down into manageable parts.

The observational equivalence $\approx$ does not enjoy the property of being a congruence with respect to the operators of CHOCS. Although it satisfies 1., 2., 4., 5. and 6. of proposition 3.4 it does not satisfy 3. which may be seen from the following counter example first presented in [Mil80]: $\tau.nil \approx nil$ but $a!p.nil + \tau.nil \not\approx a!p.nil + nil$. We may obtain a congruence using techniques presented in [Mil80] by defining $p \approx^c q$ iff $\forall C.C[p] \approx C[q]$, where C is a context. Generally a context is an expression with zero or more "holes" to be filled by an expression. We write $C[p]$ for $C[\ ]$ with p exchanged for $[\ ]$. We deliberately use the word exchange instead of substitute since according to the definition of substitution (definition 3.2) change of bound variables is taken care of, whereas free variables in p may become bound in $C[p]$.

4 Equational properties

As we saw in the previous section $\sim$ is a congruence with respect to the process constructions in CHOCS. Having established this, it naturally leads to a series of identifications of programs like: $p \mid p' \sim p' \mid p$. Of course the left hand side of this equation is a different program from that on the right hand side, but we would expect to find their behaviour equivalent, and this is in fact what the equation expresses.

The equational properties of $\sim$ may yield a better understanding of the underlying semantics of CHOCS and for the unexperienced user of the language it may turn out to be a helpful way of understanding the language and the interplay of its constructs. In the process algebraic framework the semantics of the ACP-language [BerKlo84] is given entirely as an equational theory in an algebraic setting. We shall not do so, but in fact $Pr/\sim$ may be considered as an algebra: e.g. $(Pr/\sim, +, nil)$ is an abelian monoid as justified by the first three equations of table 2.

Using these rules we may prove properties about processes without directly constructing a bisimulation. This approach is often much more manageable and the two methods may be combined when convenient. All the equational properties of table 2 are easily established by bisimulations.

We have not listed any immediate interplay between (nondeterministic) choice and parallel composition in table 2. This is due to the fact that the two operators in general do not commute, but there is a restricted interplay between them:

Proposition 4.1 *Let $\overline{x} = \{x_1 \dots x_n\}$, $\overline{y} = \{y_1 \dots y_n\}$ and $\overline{x} \cap \overline{y} \neq \emptyset$ then*

$$\begin{array}{rrcl} \textit{if} & p & = & \Sigma_i a_i?x_i.p_i + \Sigma_j a_j!p'_j.p_j \\ \textit{and} & q & = & \Sigma_k b_k?y_k.q_k + \Sigma_l b_l!q'_l.q_l \\ \textit{then} & p \mid q & \sim & \Sigma_i a_i?x_i.(p_i \mid q) + \Sigma_j a_j!p'_j.(p_j \mid q) + \\ & & & \Sigma_k b_k?y_k.(p \mid q_k) + \Sigma_l b_l!q'_l.(p \mid q_l) + \\ & & & \Sigma_{(i,l) \in \{(i,l) \mid a_i = b_l\}} \tau.(p_i[q'_l/x_i] \mid q_l) + \\ & & & \Sigma_{(j,k) \in \{(j,k) \mid a_j = b_k\}} \tau.(p_j \mid q_k[p'_j/y_k]) \end{array}$$

where $\Sigma_i \Gamma_i.p_i$ describes the sum $\Gamma_1.p_1 + \dots + \Gamma_n.p_n$ when $n > 0$ and nil if $n = 0$, knowing this notation is unambiguous because of 1. and 2. of table 2.

1. $p + p' \sim p' + p$
2. $p + (p' + p'') \sim (p + p') + p''$
3. $p + nil \sim p$
4. $p + p \sim p$
5. $p \mid p' \sim p' \mid p$
6. $p \mid (p' \mid p'') \sim (p \mid p') \mid p''$
7. $p \mid nil \sim p$
8. $p\backslash a\backslash b \sim p\backslash b\backslash a$
9. $(p + p')\backslash b \sim p\backslash b + p'\backslash b$
10. $(p\backslash b)[S] \sim p[S]\backslash S^{-1}(b)$
11. $(a?x.p)[S] \sim S(a)?x.p[S]$
12. $(a!p'.p)[S] \sim S(a)!p'.p[S]$
13. $(p + p')[S] \sim p[S] + p'[S]$
14. $(p \mid p')[S] \sim p[S] \mid p'[S]$
15. $p[S][S'] \sim p[S' \circ S]$
16. $p[Id] \sim p$
17. $(a?x.p)\backslash b \sim \begin{cases} a?x.p\backslash b & \text{if } b \neq a \\ nil & \text{otherwise} \end{cases}$
18. $(a!p'.p)\backslash b \sim \begin{cases} a!p'.p\backslash b & \text{if } b \neq a \\ nil & \text{otherwise} \end{cases}$

Table 2: Equational properties of $\sim$.

As a consequence of proposition 2.7 we know that $\approx$ satisfies the equations of table 2. Moreover $\approx$ satisfies $p \approx \tau.p$. Observational equivalence is not a congruence relation and so the equational properties of $\approx$ may be a bit uninteresting, but we conjecture that $\approx^c$ satisfies the same equational properties as $\approx^c$ of [Mil80] with the appropriate extension to take process passing into account.

In [HenMil85] and [Mil81] equations like those given in table 2 and proposition 4.1 are used to prove soundness for sets of sound and complete proof systems for the finite respectively regular sublanguages of CCS. We shall not do so in this paper since we cannot hope for a complete axiomatization of the properties of CHOCS; the reason for this will become clear in the following sections.

We have deliberately chosen to refer to the set *Names* as a set of portnames emphasizing that process values are to be thought of as communicated via ports. In any implementation of a system described in CHOCS it would be of great importance to know certain facts about these names as e.g. the number of different names, substitutivity of names etc. We may ascribe a sort (a set of portnames) to each program. To formally define the sort of a program we need a bit of notation.

Definition 4.2 *q is a derivative of p if $p \longrightarrow^* q$, where $p \longrightarrow q \equiv \exists r \in Act.p \xrightarrow{r} q$ and $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.*

Definition 4.3 *For each $L \subseteq$ Names, let Pr_L be the set of processes p such that for any derivative q of p, if $q \xrightarrow{a?q'} q''$ or $q \xrightarrow{a!q'} q''$ then $a \in L$. If $p \in Pr_L$ we say p has sort L (written $p :: L$).*

We may see how the process constructions of CHOCS act on sorts:

Proposition 4.4 1. *If $p :: L$ and $L \subseteq M$ then $p :: M$.*

2. *If $a \in L$ and $p :: L$ then $a?x.p :: L$.*
3. *If $a \in L$ and $p :: L$ then $a!p'.p :: L$ for any p'.*
4. *If $p :: L$ and $p' :: L$ then $p + p' :: L$.*
5. *If $p :: L$ and $p' :: L$ then $p \mid p' :: L$.*
6. *If $p :: L$ then $p\backslash a :: L \setminus \{a\}$.*
7. *If $p :: L$ then $p[S] :: \{S(a) \mid a \in L\}$.*

The following semantic function may be used to compute the sort of a process:

Definition 4.5 *dynamicsort : CHOCS $\rightarrow$ Names*

$$dynamicsort(p) =$$
$$\{a \in Names \mid \exists q, q', q''.p \longrightarrow^* q \xrightarrow{a?q'} q'' \text{ or } p \longrightarrow^* q \xrightarrow{a!q'} q''\}$$

This set is the minimal sort for an agent. The dynamic sort is often not convenient since it can only be determined dynamically. We are often satisfied by coarser — but easier to compute — information which may be extracted from the program text.

Definition 4.6 *We define staticsort : CHOCS $\rightarrow$ Names structurally on p:*

$$\begin{aligned}
staticsort(nil) &= \emptyset \\
staticsort(a?x.p) &= \{a\} \cup staticsort(p) \\
staticsort(a!p'.p) &= \{a\} \cup staticsort(p) \\
staticsort(p + p') &= staticsort(p) \cup staticsort(p') \\
staticsort(p \mid p') &= staticsort(p) \cup staticsort(p') \\
staticsort(p\backslash a) &= staticsort(p) \setminus \{a\} \\
staticsort(p[S]) &= \{S(a) \mid a \in staticsort(p)\} \\
staticsort(x) &= Names
\end{aligned}$$

Note how we need to "assume the worst" when encountering a variable. This is because we do not know the sort of the processes which may be substituted for the variable. In fact $a?x.x \xrightarrow{a?p} p$ for any p, and p may have any sort which is reflected both in the dynamic sort and the static sort of $a?x.x$. This "assuming the worst" resembles how static approximations of dynamic properties of sequential programming languages are made in the framework of abstract interpretation [CouCou79]. "Assuming the worst" in case of a variable implies that it is not necessary to calculate the static sort of process values as may be seen from the clause for output prefix, the sort of any process received in communication will be covered by the static assumption on variables. The information given by the static sort is often too coarse as in $p = (a?x.x \mid a!(b!nil.nil).nil)\backslash a$ since $staticsort(p) = Names\backslash\{a\}$, whereas $dynamicsort(p) = \{b\}$. As a solution to this problem one could introduce a sort declaration on each binding of variables and limit communication to processes of the prescribed sort. This would correspond to type declarations in typed programming languages like PASCAL. This is indeed the approach of [Nie88].

The dynamic sort and the static sort are of course related:

Proposition 4.7 *dynamicsort* $\subseteq$ *staticsort*

staticsort = *dynamicsort* does not hold in general as we saw above. But both *staticsort* and *dynamicsort* are sound with respect to definition 4.3 of a sort for p. The information calculated by our *staticsort* is very coarse, but the definition of sort (and types of processes in general) of [Nie88] presents a refined static sort where the sorts of processes sent and received contributes to the calculation of the sorts of processes. In general we cannot hope to show that *staticsort* = *dynamicsort* since this is undecidable, even without process passing, as a consequence of [AusBou84].

We may now give some equational properties which only hold under certain constraints on the sort.

Proposition 4.8

1. $p\backslash b \sim p$ *if* $p :: L$ *and* $b \notin L$
2. $(p \mid p')\backslash b \sim p\backslash b \mid p'\backslash b$ *if* $p :: L,\ p' :: M$ *and* $b \notin L \cap M$
3. $p\backslash b \sim p[a/b]\backslash a$ *if* $p :: L$ *and* $a \notin L$

1. shows that restriction has no effect if the restricted port does not belong to the sort of the agent. 2. shows that restriction only distributes over communication if the restriction does not involve the ports which the processes are able to communicate via. 3. shows that the name of a restricted port is not essential upto renaming. This property corresponds to the notion of α–convertibility in [EngNie86].

We have seen that almost all properties of CCS carry over to CHOCS but since CHOCS includes higher order constructs one would expect to find it more powerful and indeed it is. In CCS the recursion operator $rec\,x.p$ is the only operator capable of introducing infinite behaviours. $rec\,x.$ is a variable binder and FV and $[\ /\ ]$ have to be extended according to this (see e.g. [Mil81]). In CCS recursive processes have the following operational semantics:

$$\frac{p[rec\,x.p/x] \xrightarrow{\Gamma} p'}{rec\,x.p \xrightarrow{\Gamma} p'}$$

This inference rule basically says that a recursive process has the same derivations as its unfoldings. In CHOCS we can "program" a recursion construct to obtain infinite behaviours. To a certain extent this construct resembles the Y combinator in the λ–calculus.

Definition 4.9 *Let* $Y_x[\,]$ *be the following context:*

$$(a?x.([\,] \mid a!x.nil) \mid a!(a?x.([\,] \mid a!x.nil)).nil)\backslash a$$

Theorem 4.10 *if* $p :: L$ *and* $a \notin L$ *then* $Y_x[p] \sim rec\,x.\tau.p$

PROOF: The proof of this theorem needs the following property of substitution:

$$\text{if } x \neq y \text{ then } p[p'/x][p''/y] \equiv p[p''/y][p'[p''/y]/x]$$

which is easily established by structural induction on p. Then the relation:

$$\begin{aligned} R \;=\; & \{(q[rec\,x.\tau.p/x], (q[a?x.(p \mid a!x.nil)/x] \mid \\ & a!(a?x.(p \mid a!x.nil)).nil \mid \underbrace{nil \mid \ldots \mid nil}_{m})\backslash a) \\ & \mid\; FV(q) \subseteq \{x\} \text{ and } m \geq 0\} \end{aligned}$$

is a bisimulation. This may be shown by induction on the number of inferences used to establish any transition of q observing the structure of q. In the case where q has the form $a?y.q'$ or $rec\,y.q'$ we need the above property of substitution. The theorem then follows by choosing $q \equiv x$. (The proof follows the pattern of the proof of proposition 4.6 of [Mil83]). $\square$

This proof is limited to the case where at most x is free in q. The extension to the case where there are other free variables is routine, but demands for a definition of higher order bisimulation for open terms (which is standard and straightforward).

$rec\,x.p$ is not derivable in the sense of [Pra88] but it can be closely mimicked by $Y_x[p]$ and $Y_x[p] \approx rec\,x.p$, where $\approx$ is observational equivalence.

Example 4.11 *Let* $p \equiv b!.x$ *then according to the inference rules of definition 3.3* $Y_x[p]$ *has the following derivations:*

$$\begin{gathered} Y_x[p] \equiv (a?x.(b!.x \mid a!x.nil) \mid a!(a?x.(b!.x \mid a!x.nil)).nil)\backslash a \\ \downarrow_{\tau} \\ (b!.(a?x.(b!.x \mid a!x.nil) \mid a!(a?x.(b!.x \mid a!x.nil)).nil \mid nil)\backslash a \\ \downarrow_{b!} \\ (a?x.(b!.x \mid a!x.nil) \mid a!(a?x.(b!.x \mid a!x.nil)).nil \mid nil)\backslash a \\ \downarrow_{\tau} \\ (b!.(a?x.(b!.x \mid a!x.nil) \mid a!(a?x.(b!.x \mid a!x.nil)).nil) \mid nil \mid \\ nil \mid nil)\backslash a \\ \downarrow_{b!} \\ \vdots \end{gathered}$$

Note how $Y_x[\,]$ *needs a* τ*–transition to unwind the "recursion". This resembles the unwinding of recursion in the inference rule of recursion in TCCS [dNiHen87]:* $rec\,x.p \rightsquigarrow p[rec\,x.p/x]$, *where* $\rightsquigarrow$ *may be read as* $\xrightarrow{\tau}$.

5 CHOCS and the λ–calculus

CCS is a powerful language; It is capable of expressing all Turing definable functions by encoding of Turing machines (see [Mil83]). Since CCS is a sublanguage of CHOCS this must be true for CHOCS as well. But the nature of CHOCS is much closer to the λ–calculus and in this section we shall see how the untyped λ–calculus may be encoded in CHOCS.

First we recall the syntax of the untyped λ–calculus as described in [Bar85]:

Definition 5.1 *The set of* λ*–terms* Λ *is defined inductively as follows:*

1. $x \in \Lambda$
2. $M \in \Lambda \Rightarrow (\lambda x.M) \in \Lambda$

3. $M, N \in \Lambda \Rightarrow (M\ N) \in \Lambda$

where $x \in V$ (a set of variables).

This language consists of variables, function abstraction and function application. The λ–calculus has a rich theory as documented in e.g. [Bar85], consisting of concepts such as conversion, reduction, theories and models.

A simple translation of the λ–calculus may be given by the following definition:

Definition 5.2 *We define $[\![\ ]\!] : \Lambda \to CHOCS$ structurally:*

1. $[\![x]\!] = x$
2. $[\![\lambda x.M]\!] = i?x.i![\![M]\!].nil$
3. $[\![M\ N]\!] = ([\![M]\!][o/i] \mid o![\![N]\!].o?x.x)\backslash o$

Note that for any $M \in \Lambda$: $[\![M]\!] :: \{i\}$ and that application only needs two communication channels. Since the function $[\![\ ]\!] : \Lambda \to CHOCS$ has no additional arguments we may view it as a definition of a set of derived operators in CHOCS. Clause 3. shows how we may view parallel composition as a generalization of function application. However, we need a rather elaborate protocol to ensure that we do not mix arguments in applications and we therefore feed the arguments sequentially. A tempting definition of the clause for application is $[\![M\ N]\!] = ([\![M]\!] \mid i![\![N]\!].nil)\backslash i$. Unfortunately this definition does not work since the restriction $\backslash i$ prevents application to other arguments as in e.g. $M\ N\ N'$. A different approach is presented in [Bou88] where a special operator takes care of this problem. The cost of this is a complication of the definition of equivalence between processes.

In the following we shall see that some of the most interesting properties of the λ–calculus are carried over via the translation. First we make clear the connection between substitution in the λ–calculus and in CHOCS.

Lemma 5.3

$$[\![M[x := N]]\!] \equiv [\![M]\!][[\![N]\!]/x]$$

where $[x := \ ]$ is substitution in the λ–calculus as defined in definition 2.1.15 of [Bar85] and $[\ /x]$ is substitution in CHOCS according to definition 3.2.

Proof: By structural induction on M. □

Using this lemma we may show that β–conversion in the λ–calculus is "preserved" by the translation:

Proposition 5.4

$$[\![(\lambda x.M)N]\!] \approx [\![M[x := N]]\!]$$

Proof: We demonstrate how the left hand side of this equation may do an initial series of internal τ–moves to a process equivalent to the right hand side.

$$\begin{array}{c} [\![(\lambda x.M)N]\!] = ((i?x.i![\![M]\!].nil)[o/i] \mid o![\![N]\!].o?x.x)\backslash o \\ \downarrow_\tau \\ ((i!([\![M]\!][[\![N]\!]/x]).nil)[o/i] \mid o?x.x)\backslash o \\ \downarrow_\tau \\ (nil[o/i] \mid ([\![M]\!][[\![N]\!]/x]))\backslash o \\ \sim \\ [\![M]\!][[\![N]\!]/x] \end{array}$$

Since $[\![M]\!] :: \{i\}$ for all $M \in \Lambda$ we may use the properties of table 2 and proposition 4.8 to infer the conclusion of this proposition. □

The theory λ as presented in [Bar85] is strongly related to the translation $[\![\ ]\!] : \Lambda \to CHOCS$ and the properties of CHOCS:

Theorem 5.5 *if $\lambda \vdash M = N$ then $[\![M]\!] \approx [\![N]\!]$.*

Proof: By structure of $\lambda \vdash M = N$ □

The converse does not hold in general, but $\approx$ induces an equality relation on Λ and it is straightforward to verify that the relation $R = \{(M,N) \mid [\![M]\!] \approx [\![N]\!],\ M,N \in \Lambda\}$ is a compatible congruence relation. Proposition 5.4 shows that $\beta = \{((\lambda x.M)N, M[x := N]) \mid M,N \in \Lambda\} \subseteq R$ and therefore $=_\beta \subseteq =_R$. The notion of β–equality is important, but in the standard theory of the λ–calculus it is the notion of head normal form, based on Böhm trees, which yields the meaning of a λ–term. Terms without a head normal form are identified. But $=_R$ distinguishes more λ–terms than the standard theory since e.g. $[\![\lambda x.\Omega]\!] \not\approx [\![\Omega]\!]$, where $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$. In general we do not have the full η–conversion i.e. $\lambda \vdash \lambda x.M\ x = M$ if $x \notin FV(M)$ but if M has the form $\lambda y.M'$ we have $[\![\lambda x.(\lambda y.M')\ x]\!] \approx [\![\lambda x.M[y := x]]\!] \approx [\![M]\!]$ which is easy to establish using the properties of table 2 and proposition 5.4. This restricted version of η–conversion is close to the restricted version valid in the Lazy–λ–calculus of [Abr88]. Furthermore connections to the Lazy–λ–calculus are strengthened by the following proposition:

Proposition 5.6 $[\![\Omega]\!] \sim rec\,x.\tau.x \sim Y_x[x]$

This shows that the standard unsolvable term Ω of the λ–calculus yields a divergent process in CHOCS, i.e. a process only capable of performing an infinite series of internal moves. Whether the above suggested connections to the Lazy–λ–calculus can be further strengthened is an open question being investigated for the moment.

6 CHOCS as a metalanguage

In this section we shall study how CHOCS may be used as a metalanguage in the definition of the semantics of programming languages. The study is a case analysis of a simple imperative toy language, called P, first studied in [Mil80]. The meaning of the language P is given in a phrase–by–phrase style resembling denotational language definitions though we shall not give any semantic domains. The language P is devised in such a way that a programmer is partly protected from unwanted deadlocks. This is obtained through a disciplined way of communication between components sharing some resources. In [Mil80] R. Milner points out the difficulties of describing procedures in P using CCS. It is not obvious that CCS or the extension of CCS justified by the developments in [Mil83] can describe concurrent procedure invocations satisfactory. In [EngNie86] U. Engberg and M. Nielsen show how CCS with labels as first class objects may be used to describe concurrent procedure invocations, unfortunately their solution is very complicated and it does not look like procedure descriptions of sequential programming languages. We show how procedures in P may be handled straightforward in a way resembling how procedures in sequential imperative languages are handled in denotational descriptions based on the λ–calculus. Most of the definitions not concerning procedures may be found in [Mil80], but for the sake of completeness we present the full language definition.

To allow for other values than process values in CHOCS we use the technique of [Mil83] and introduce a D–indexed family of actions $a?_d$, $a!_d$, $d \in D$ for each value domain D. Due to the fact that only finite sums of processes can be handled in the version of CHOCS presented in this paper we restrict our attention to finite value domains as e.g. the set of booleans and finite subsets of the integers. We let $\alpha?_x.p$ abbreviate $\Sigma_{d \in D}\alpha?_d.p\{d/x\}$ where $\{d/x\}$ means exchanging all occurrences of x in p by d as e.g. $\alpha?_x.\beta!_x.nil\{d/x\} \equiv \Sigma_{d \in D}\alpha?_d.\beta!_d.nil$. We shall use the following construct from [Mil83]: If b is a boolean valued expression in x then let $\alpha?_x.(if\ b\ then\ p\ else\ p')$ be encoded by $\Sigma_{x \in D \& b}\alpha?_x.p + \Sigma_{x \in D \& \neg b}\alpha?_x.p'$. We should not confuse $\alpha?_x.p$ with $\alpha?x.p$ since the first is a convenient short hand notation and the latter is part of the CHOCS syntax.

The toy language P

Programs of P are built from declarations D, expressions E and commands C, using assignments to program variables X. Some set of functions F is assumed and for the cause of simplicity we do not consider types of expressions. P has the following abstract syntax:

```
Declarations:
D ::= var X | D; D | proc P(value X, result X') is C
Expressions:
E ::= X | F(E_1, ..., E_n)
Commands:
C ::= X := E | C; C | if E then C else C' |
while E do C | C par C' | input X | output E |
skip | begin D; C end | call P(E,X)
```

Table 3: Syntax of P

In the study of concurrent programming languages a question of interest is how to evaluate programs like:

```
begin
var X;
X := 0;
(X := X + 1 par X := X + 1);
output X
end
```

The semantics presented here will yield the answers 1 or 2. Readers are referred to [Mil80] for a discussion of an alternative specification to rule out the answer 1.

To give a smooth definition of the semantics of P we need some auxiliary definitions.

To each variable X we associate a register Reg_X. Generally it follows the following pattern:

$$Loc = \alpha?_x.Reg(x)$$

$$Reg(y) = \alpha?_x.Reg(x) + \gamma!_y.Reg(y)$$

and thus for X we will have $Loc_X = Loc[\alpha_X\ \gamma_X/\alpha\ \gamma]$. Initially we write in a value, thereupon we can read this value on γ or overwrite the contents of Loc via α. We have written the above definition in an equation style to make it more readable. The proper CHOCS definition is: $Loc = (\alpha?_x.h!_x.nil \mid Reg)\backslash h$ where $Reg = Y_{Reg}[h?_x.(\alpha?_x.h!_x.Reg + \gamma!_x.h!_x.Reg)] \mid Y_{Keep}[h?_x.h!_x.Keep]$. The second component of this process takes care of the parameters in the recursion of the above equations. (This is in fact a general technique for simulating the parameterized recursion of [Mil83]). We also associate a register to each procedure P. It may be defined in the same way as above with x substituted with x.

To each n–ary function symbol F we associate a function f which is represented by:

$$b_f = \rho_1?_{x_1}.\cdots\rho_n?_{x_n}.\rho!_{f(x_1 \ldots x_n)}.nil$$

Constants will thus be represented as e.g. $b_{true} = \rho!_{true}.nil$ The result of evaluating an expression is always communicated via ρ. It is therefore useful to define:

$$p\ result\ p' = (p \mid p')\backslash\rho$$

We shall adopt the protocol of signalizing successful termination of commands via δ and it therefore convenient to define:

$$done = \delta!.nil$$

$$p\ before\ p' = (p[\beta/\delta] \mid \beta?.p')\backslash\beta \quad (\beta\ new)$$

$$p\ par\ p' =$$

$$(p[\delta_1/\delta] \mid p'[\delta_1/\delta] \mid \delta_1?.\delta_2?.done + \delta_2?.\delta_1?.done)\backslash\delta_1\backslash\delta_2$$

$$(\delta_1\delta_2\ new)$$

We may now give the semantics of P by the translation into CHOCS shown in table 4. In the equation for [begin D; C end] we let $\backslash L_D$ abbreviate restriction with respect to α and γ channels for all variables and procedures declared in D. The procedure definition creates a location to store the procedure process. The restriction $\backslash\alpha_P$ ensures that this process cannot be overwritten after the definition. The procedure process needs two locations, one for each parameter. These locations are kept local by the restrictions $\backslash\alpha_X\backslash\alpha_Y\backslash\gamma_X\backslash\gamma_Y$. To ensure static binding of variables in a procedure body the procedure process is embraced by a renaming of all read and write signals to variable locations. This is done simply by tagging the signals with the name of the procedure. The tagged signals are able to escape the restriction $\backslash L_D$ of any block except the block where the procedure is defined. The $Transform$ process, located in the block where the procedure is defined, transforms the tagged signals back to untagged read and write signals. These will of course effect the variable locations in this environment. The value to the value parameter is communicated via α_{P_v} and the result of the procedure is communicated via γ_{P_v}.

Declarations:

$$\begin{aligned}
\llbracket \texttt{var}\ X \rrbracket &= \mathit{Loc}_X \\
\llbracket D; D' \rrbracket &= \llbracket D \rrbracket \mid \llbracket D' \rrbracket \\
\llbracket \texttt{proc}\ P(\texttt{value}\ X,\ \texttt{result}\ Y\ \texttt{is}\ C \rrbracket &= ((\mathit{Loc}_P \mid \alpha_P!((\ \textit{procedure process}\ ).\mathit{nil}))\backslash\alpha_P) \mid \mathit{Transform}
\end{aligned}$$

where *procedure process* $=$

$$(((\alpha_{P_v}?_x.\alpha_X!_x.\mathit{done})\ \mathit{before}\ \llbracket C \rrbracket\ \mathit{before}\ (\gamma_Y?_x.\gamma_{P_v}!_x.\mathit{done}) \mid \mathit{Loc}_X \mid \mathit{Loc}_Y)\backslash\alpha_X\backslash\alpha_Y\backslash\gamma_X\backslash\gamma_Y)[\alpha^P_{X'}\ \gamma^P_{X'}/\alpha_{X'}\ \gamma_{X'}]$$

and $\mathit{Transform} = Y_{Tran}[\Sigma_{X'}\alpha^P_{X'}?_x.\alpha_{X'}!_x.\mathit{Tran} + \Sigma_{X'}\gamma_{X'}?_x.(\gamma^P_{X'}!_x.\mathit{Tran} + \mathit{Tran})]$

Expressions:

$$\begin{aligned}
\llbracket X \rrbracket &= \gamma_X?_x.\rho!_x.\mathit{nil} \\
\llbracket F(E_1,\ldots,E_n) \rrbracket &= (\llbracket E_1 \rrbracket[\rho_1/\rho] \mid \ldots \mid \llbracket E_n \rrbracket[\rho_n/\rho] \mid b_f)\backslash\rho_1\ldots\backslash\rho_n
\end{aligned}$$

Commands:

$$\begin{aligned}
\llbracket X := E \rrbracket &= \llbracket E \rrbracket\ \mathit{result}\ (\rho?_x.\alpha_X?_x.\mathit{done}) \\
\llbracket C; C' \rrbracket &= \llbracket C \rrbracket\ \mathit{before}\ \llbracket C' \rrbracket \\
\llbracket \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C' \rrbracket &= \llbracket E \rrbracket\ \mathit{result}\ \rho?_x.(\mathit{if}\ x\ \mathit{then}\ \llbracket C \rrbracket\ \mathit{else}\ \llbracket C' \rrbracket) \\
\llbracket \texttt{while}\ E\ \texttt{do}\ C \rrbracket &= Y_w[\llbracket E \rrbracket\ \mathit{result}\ \rho?_x.(\mathit{if}\ x\ \mathit{then}\ (\llbracket C \rrbracket\ \mathit{before}\ w)\ \mathit{else}\ \mathit{done})] \\
\llbracket C\ \texttt{par}\ C' \rrbracket &= \llbracket C \rrbracket\ \mathit{par}\ \llbracket C' \rrbracket \\
\llbracket \texttt{input}\ X \rrbracket &= \iota?_x.\alpha_X!_x.\mathit{done} \\
\llbracket \texttt{output}\ E \rrbracket &= \llbracket E \rrbracket\ \mathit{result}\ (\rho?_x.o!_x.\mathit{done}) \\
\llbracket \texttt{skip} \rrbracket &= \mathit{done} \\
\llbracket \texttt{begin}\ D; C\ \texttt{end} \rrbracket &= (\llbracket D \rrbracket \mid \llbracket C \rrbracket)\backslash L_D \\
\llbracket \texttt{call}\ P(E,Z) \rrbracket &= \llbracket E \rrbracket\ \mathit{result}\ ((\rho?_x.\alpha_{P_v}!_x.\mathit{done}) \\
&\qquad \mathit{par}\ (\gamma_P?x.x)\ \mathit{par}\ (\gamma_{P_v}?._x.\alpha_Z!_x.\mathit{done}))\backslash\alpha_{P_v}\backslash\gamma_{P_v}
\end{aligned}$$

Table 4: Semantics of P

These signals are not affected by the embracing renaming. The procedure call first evaluates the argument then reads the location Loc_P to get a copy of the procedure process. Note how each procedure process is self–contained with local environments for the parameters. If a recursive call to the procedure P occurs in the body C a new copy of the procedure process will be obtained. This is true for concurrent activations of the same procedure as well and we have:

$$\begin{gathered}
\llbracket \texttt{begin proc}\ P(\texttt{value}\ X,\ \texttt{result}\ Y)\ \texttt{is}\ C; \\
\texttt{call}\ P(E,Z)\ \texttt{par call}\ P(E',Z')\ \texttt{end} \rrbracket \\
\approx \\
\llbracket \texttt{begin var}\ X; \texttt{var}\ Y; X := E; C; Z := Y\ \texttt{end} \\
\texttt{par begin var}\ X; \texttt{var}\ Y; X := E'; C; Z := Y'\ \texttt{end} \rrbracket
\end{gathered}$$

which may be verified by expanding the semantic clauses.

Another common parameter mechanism used in imperative programming languages is the call by reference mechanism. This mechanism may be modelled in CHOCS by the following semantic definitions:

$$\llbracket \texttt{proc}\ P(\texttt{ref}\ X)\ \texttt{is}\ C \rrbracket =$$
$$((\mathit{Loc}_P \mid \alpha_P!(\ \textit{procedure process}\ ).\mathit{nil})\backslash\alpha_P) \mid \mathit{Transform}$$

where
procedure process $= \llbracket C \rrbracket[\alpha_{P_v}\ \gamma_{P_v}/\alpha_X\ \gamma_X][\alpha^P_{X'}\ \gamma^P_{X'}/\alpha_{X'}\ \gamma_{X'}]$.

$$\llbracket \texttt{call}\ P(Y) \rrbracket = \gamma_P?x.(x[\alpha_Y\ \gamma_Y/\alpha_{P_v}\ \gamma_{P_v}])$$

Note how this parameter mechanism works; We just link the register associated with the parameter in the call with the procedure process via renaming. This is obtained by the inner renaming in the procedure body which ensures that read and write signals to the formal parameter escape the outer renaming. This has the effect that they are linked to the actual parameter in the calling environment.

7 Concluding Remarks

The denotational approaches of [KenSle88] and [Chr88] are very complicated. Both are formulated in a category theoretical setting and the main purpose of both papers are to establish functors describing the properties of process passing. A lot of effort is put into assuring that these functors can be used together with standard domain constructors and in recursive domain equations. We believe that it is not necessary to establish special functors for this purpose and that standard domain theory is sufficient to give denotational semantics for languages with processes as first class objects. In [Abr87a] S. Abramsky shows how it is possible to give both an operational semantics and a denotational semantics together with a logic for description of properties and to link all three approaches together in a unifying framework. This framework is used in [Abr87b] by applying methods of denotational semantics to problems concerning nondeterminism and concurrency, and it is shown how Hennessy–Milner Logic (HML), the Plotkin power domain and labeled transition systems can be unified in the description of nondeterministic and concurrent processes. Our preliminary investigations show that there is an extension of HML characterizing the extended version of bisimulation given in definition 2.2. If we extend the transition systems used as the underlying semantics for CHOCS with a divergence predicate in the style of [Wal88], it is possible to relate these with the semantic domain $D \cong \mathcal{P}^0(\Sigma_{a\in Act} D \times D + \Sigma_{a\in Act} D \times D)$, where $\mathcal{P}^0$ is the Plotkin power domain with the empty set augmented and $\Sigma_{a\in Act}$ is generalized coproduct. The partial order of this domain coincides with higher order bisimulation extended to take divergence into account, again following the ideas of [Wal88]. It is a topic for further research if this domain and the extended version of HML can be unified in the framework of [Abr87a].

Another direction for further research is how the notion of processes as first class objects relates to the modelling of dynamic use of communication channels as suggested in [Mil83b] or the more refined treatment of portnames as first class objects in ECCS as described in [EngNie86]. Our preliminary investigations point in the direction that the two approaches are expressively equivalent. It is possible to go from CHOCS to ECCS using a technique resembling implementing functional languages in imperative languages with GOTO statements and one can go from ECCS to CHOCS using a kind of continuation style semantics. However, further investigations into the correctness of the two approaches are necessary. We feel that ECCS is much closer to a machine level or assembly language level of computations of dynamic process communication and that one should consider CHOCS as a specification language abstracting away the complicated nature of the dynamic use of communication channels, analogous to using procedures as a clean way of using GOTO statements in imperative programming languages.

CHOCS as a metalanguage and a specification language in general has to be tried out on many more examples before any conclusions of its usability can be drawn, but as the example given in this paper suggests it is expressively powerful and it has shown to be of theoretical interest. Application of CHOCS to more "real life" concurrent systems will show how these systems can be treated and hopefully demonstrate that these can be treated as cleanly as the treatment of the λ-calculus and the toy language P.

Acknowledgement

I thank the referees for pointing me in the direction of the work by D. Austry and G. Boudol and for making me aware of the fact that my first definition of procedures in the semantics of P described dynamic binding. I would also like to thank everybody who has commented on previous informal presentations of this material; This includes comments by S. Abramsky, J. Andersen, S. Brookes, M. Dam, K. G. Larsen, L. Leth, R. Milner, F. Nielson, L. Ong, I. Phillips, A. Skou, B. Steffen and A. Stoughton.
I am grateful to The Danish Natural Science Research Council; The Danish Research Academy; Institute of Mathematics, Århus University and Department of Mathematics and Computer Science, Aalborg University Centre for providing me with fundings for my research.

References

[Abr87a] S. Abramsky: *Domain Theory in Logical Form*, Proceedings of LICS 87. Full version submitted to Annals of Pure and Applied Logic, 1987.

[Abr87b] S. Abramsky: *A Domain Equation For Bisimulation*, Unpublished manuscript, Dept. Computing, Imperial College, London 1987.

[Abr88] S. Abramsky: *The Lazy Lambda Calculus*, to appear in Declarative Prg. ed. D. Turner, Addison Wesley, 1988.

[AusBou84] D. Austry & G. Boudol: *Algèbre de Processus et Synchronisation*, Theoretical Computer Science 30(1) pp. 91–131, 1984.

[AstReg87] E. Astesiano & G. Reggio: *SMoLS-Driven Concurrent Calculi*, Proceeding from TAPSOFT 87, Pisa, LNCS 249, Springer Verlag, 1987.

[Bar85] H. P. Barendreght: *The Lambda Calculus — Its Syntax and Semantics*, North-Holland 1985.

[BerKlo84] J. Bergstra & J. W. Klop: *Process Algebra for Synchronous Communication*, Information and Control 60, pp. 109–137, 1984.

[Bou88] G. Boudol: *Towards a Lambda-Calculus for Concurrent and Communicating Systems*, Research Report nr. 885, INRIA Sophia Antipolis, Autumn 1988.

[Chr88] P. Christensen: *The Domain of CSP Processes*, incomplete draft, The Technical University of Denmark, 1988.

[CouCou79] P. Cousot & R. Cousot: *Systematic design of Program Analysis Frameworks*, In: Conf. Record of the 6th ACM symposium on Principles of Programming Languages 1979.

[dNiHen87] M. Hennessy & R. de Nicola: *CCS without τ's*, Proceeding from TAPSOFT 87, Pisa, LNCS 249, Springer Verlag, 1987.

[EngNie86] U. Engberg & M. Nielsen: *A Calculus of Communicating Systems with Label Passing*, DAIMI PB-208 Aarhus University Computer Science Department, 1986.

[HenMil85] M. Hennessy & R. Milner: *Algebraic Laws for Nondeterminism and Concurrency*, Journal of the Association for Computing Machinery, pp. 137-161, 1985.

[KenSle83] J. R. Kennaway & M. R. Sleep: *Syntax and Informal Semantics of DyNe, a Parallel language*, LNCS 207, Springer Verlag, 1983.

[KenSle88] J. R. Kennaway & M. R. Sleep: *A Denotational Semantics for First Class Processes*, Draft, School of Information Systems, University of East Anglia, Norwich, U.K., August 1988.

[Mil80] R. Milner: *A Calculus of Communicating Systems*, LNCS 92, Springer Verlag, 1980.

[Mil81] R. Milner: *A Complete Inference System for a Class of Regular Behaviours*, University of Edinburgh 1981.

[Mil83] R. Milner: *Calculi for Synchrony and Asynchrony*, Theoretical Computer Science 25 (1983), pp 269-310, North Holland.

[Mil83b] R. Milner: *Parallel Combinator Reduction Machine*, LNCS 207, Springer Verlag, 1983.

[Nie88] F. Nielson: *The Typed λ-Calculus with First-Class Processes*, Technical Report ID-TR: 1988-43 ISSN 0902-2821, Department of Computer Science, Technical University of Denmark, August 1988.

[Par81] D. Park: *Concurrency and Automata on Infinite Sequences*, LNCS 104, Springer Verlag, 1981.

[Plo81] G. Plotkin: *A Structural Approach to Operational Semantics*, DAIMI FN-19 Aarhus University Computer Science Department 1981.

[Pra88] K. V. S. Prasad: *Combinators and Bisimulation Proofs for Restartable Systems*, Ph. D Thesis, Edinburgh University, 1988.

[Tar55] A. Tarski: *A Lattice-Theoretical Fixpoint Theorem and Its Applications*, Pacific Journal of Math. 5, 1955.

[Wal88] D. Walker: *Bisimulation and Divergence*, Proc. of LICS 88, ISBN 0-8186-0853-6.