A Calculus of Communicating Systems with Label Passing

Uffe Engberg
Mogens Nielsen

May 1986[*]

[*]This is the original MacWrite document converted to LaTeX. However, a few typos have been removed and the formatting adapted to the LaTeX-style.

Contents

Preface

This report is essential Uffe Engbergs thesis for the MSc degree from Department of Computer Science, Aarhus University—except that all proofs of theorems have been left out. Should anyone have interest in particular proofs, they may be obtained by contacting one of the authors.

1 Introduction

In the original version of CCS, as presented in [Mil1], *structured* dynamically evolving configurations such as the pushdown store can be obtained by means of recursion and the chaining combinator. It is less clear thar the same can be obtained for *unstructured* dynamically evolving configurations like the example studied in [Mil1, chapter 9], which was the natural one of allowing unboundedly many concurrent activations of a single procedure in a concurrent programming language. There it was pointed out that a solution would be to allow the passage of communication links as values between one agent and another, but that CCS probably was defective in this respect. It was also noted that it's usefulness was not limited to language translations. In [Dog] it is mentioned that in general, the exchange of ports (communication links) between agents, would be a natural way to model the exchange of communication capabilities.

In later versions of CCS (see [Mil2] and [Mil3]) a more basic calculus, which allow infinite summation but not direct value communication, were introduced. It was shown how the original version - a richer calculus - could be encoded. Value communication and manipulation was encoded essentially by indexing the labels and the agent identifiers. Labels or communication links could also be encoded (as special cases of values). A similar approach has been made by [AsZ] which conceptually only differs a little from Milners approach. A very different called LNET is presented in [KeS]. LNET might be described as a hybrid of actor languages and CCS. In spite of this we have made a different approach for several reasons. Although the later basic calculus in a sense allows the passage of communication links as values between one agent and another Milner himself notices in [Mil3]: "It is quite certain that the slender syntax of our basic calculus and even the derived notations which we have considered, are not sufficient always to present such applications [with passage of communication links] in a *lucid* way". Most of the problems are left for the "programmer".

Our approach will be more in keeping with the original version of CCS and at the same time widen the connection to the lambda-calculus and reduce the number of primitive operators (no relabelling) without loss of expressiveness. In [Mil1] Milner ask the question whether CCS's primitive constructs are the smallest possible set and says that they need a re-examination. Since we have not got the relabelling operator we thereby to some extend deal with this question. It is our belief that the parts of our approach which concerns this could be done for the basic calculus too. We will now discuss what requirements the new calculus allowing passage of communication links should meet. It will be referred to as ECCS (Extended CCS).

In what follows there is a slight syntactical difference to CCS. $\alpha?x.$— is written for $\alpha x.$—, where x is *bound by* ? and it's scope is — meaning that a value can be received at α. Similar we write $\alpha!v.B$ for sending a value.

To get a first idea of what we mean by allowing passage of communication links (labels) consider the example:

$$B_0 \mid B_1 = \alpha!8.\beta?x.\delta!x.\text{nil} \mid \alpha?y.\beta!y.\text{nil}$$

which is a CCS program. It can according to CCS develop like:

$B_0 \mid B_1$
$\downarrow^{\tau}$
$\beta?x.\delta!x.\text{nil} \mid \beta!8.\text{nil}$
$\downarrow^{\tau}$
$\delta!8.\text{nil} \mid \text{nil}$

If we replace 8 by λ it is no longer a CCS program, but we wish such communications of labels to be possible in ECCS. If x and y are replaced by x and y - variables qualifying over labels - we expect the program to be able to develop in the same way:

$\alpha!\lambda.\beta?\mathsf{x}.\delta!\mathsf{x}.\text{nil} \mid \alpha?\mathsf{y}.\beta!\mathsf{y}.\text{nil}$
$\downarrow^{\tau}$
$\beta?\mathsf{x}.\delta!\mathsf{x}.\text{nil} \mid \beta!\lambda.\text{nil}$
$\downarrow^{\tau}$
$\delta!\lambda.\text{nil} \mid \text{nil}$

Communication of labels would be of no use if it was not *possible to use a received label for later communication*, so the following modification of the example:

$$B_0 \mid B_1 = \alpha!\lambda.\lambda?x.\delta!x.\text{nil} \mid \alpha?\mathsf{y}.\mathsf{y}!5.\text{nil}$$

should be possible such that a development could be:

$B_0 \mid B_1$
$\downarrow^{\tau}$
$\lambda?x.\delta!x.\text{nil} \mid \lambda!5.\text{nil}$
$\downarrow^{\tau}$
$\delta!5.\text{nil} \mid \text{nil}$

Up till now there has probably not been any problems to understand these basic requirements. This is due to the simplicity of the examples. In [Dog] the following more complicated example is studied:

$$(B_0 \mid B_1) \setminus \alpha \mid B_2 = (\lambda ? y . y ! . \mathrm{nil} \mid \lambda ! \alpha . \alpha ? . \mathrm{nil}) \setminus \alpha \mid \lambda ! \alpha . \alpha ? . \mathrm{nil}$$

The agent B_0 can receive a label at λ and the label is bound to the variable y. If the received label α comes from the agent B_1 it agrees with our intuition if the system upon the communication results in:

$$(\alpha ! . \mathrm{nil} \mid \alpha ? . \mathrm{nil}) \setminus \alpha \mid \lambda ! \alpha . \alpha ? . \mathrm{nil}.$$

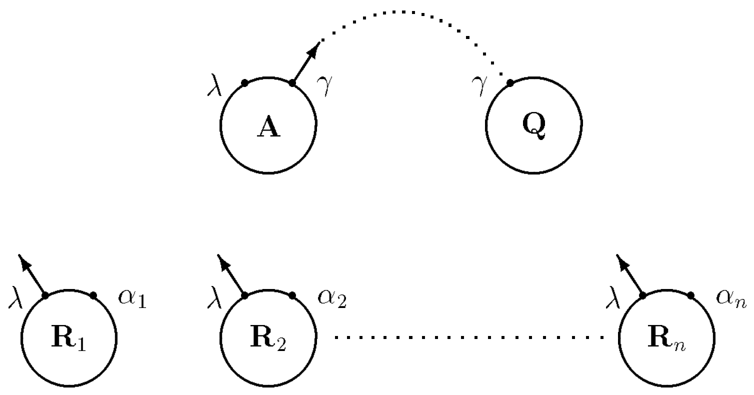
But what should the result look like if the label received originates from the outside agent B_2? Should it be possible to pass α from B_2 to B_0? If the system instead looked like:

$$(B_0 \mid B_3) \setminus \beta \mid B_2 = (\lambda ? y . y ! . \mathrm{nil} \mid \lambda ! \beta . \beta ? . \mathrm{nil}) \setminus \beta \mid \lambda ! \alpha . \alpha ? . \mathrm{nil}, \alpha \neq \beta$$

it would seem natural if the result was: $(\alpha ! . \mathrm{nil} \mid B_3) \setminus \beta \mid \alpha ? . \mathrm{nil}$. The labels visible for agent B_2 are the same in both cases. From the viewpoint of B_2 there seems no reason why $(B_0 \mid B_1) \setminus \alpha$ and $(B_0 \mid B_3) \setminus \beta$ should behave differently. It will therefore be a *central requirement to ECCS, that the name of a label restricted shall be of no importance to the behaviour* in the same way as change of bound variable in $\lambda ? x .$— does not influence the behaviour in CCS. This also seems natural if one takes up the the attitude that one is communicating via links and that the communications via a certain link should be the same no matter what name is chosen for that link. In terms of experiments on machines as sketched in [Mil1]: the buttons are the same no matter what name is printed on them.

The same question rises in a different situation and the problems seems closely connected. Let $\lambda ? y . B = \lambda ? y . (\alpha ! 5 . \mathrm{nil} \mid \alpha ? x . y ! x . \mathrm{nil}) \setminus \alpha$. What should the result look like after a label is received at λ and substituted for y in B? The situation is very similar to the one above, except for dependence of the names of the restricted labels is displaced to the substitution. We therefore *demand the same independence of actual names used for restriction when substituting a label.*

We will now study one further requirement to ECCS through an example mentioned in [Mil3]. Consider the agent $\mathbf{A}$ managing some resources: $\mathbf{R}_i$ $(1 \leq i \leq n)$ which signals to $\mathbf{A}$ via λ when they are available. Other agents makes requests for resources to $\mathbf{A}$ via γ. Let $\mathbf{Q}$ be such an agent potentially requesting. The situation can be pictured as:



where the resource $\mathbf{R}_i$ is accessed through α_i. A common solution is to write the system as

$$(\mathbf{Q} \mid \mathbf{A} \mid \mathbf{R}_1 \mid \ldots \mid \mathbf{R}_n) \setminus A, \text{ where } A = \{\alpha_i; 1 \le i \le n\}$$

and to let $\mathbf{A}$ somewhere contain a subexpression like $\gamma!i.B$ meaning that $\mathbf{A}$ communicate the index of an available resource via γ; and let $\mathbf{Q}$ contain a subexpression like $\gamma?x.(\ldots\alpha_x!\ldots\alpha_x?\ldots)$ where $\gamma?k$ means receiving the index and using it for communication with resource $\mathbf{R}_x$ via α_x. The problem of such solutions passing indexes as a kind of identification is that all potential resources must be known at the time when the administrator $\mathbf{A}$ and the system is written in order not to mix up indexes. Furthermore the family of indexed labels $\{\alpha_i\}$ must be known in advance. To illustrate this we consider a very simple system with a requesting agent and two resources.

$$\begin{aligned}
\mathbf{Q} &= \gamma?x.\alpha_x!8.\alpha_x?y.\delta!y.\mathrm{nil} \\
\mathbf{R}_1 &= \mathrm{fix}\, X\langle\lambda!1.\alpha_1?x.\alpha_1!2 * x.X\rangle \\
\mathbf{R}_2 &= \mathrm{fix}\, X\langle\lambda!2.\alpha_2?x.\alpha_2!x + x.X\rangle
\end{aligned}$$

We leave out the details of $\mathbf{A}$. A would in this example be $\{\alpha_1, \alpha_2\}$.

If one wants to add a new resource $\mathbf{R}_3$ to the system one must inspect the system to see that communications between requesting agents is done via labels of type α_x and that α_1, α_2 already is used. It is not enough to know the way they communicate (the communication protocol they use) and that requests for resources are done to $\mathbf{A}$ via γ and resources availability is reported via λ. Furthermore $\mathbf{A}$ must be extended with $\{\alpha_3\}$ if '3' is used to identify $\mathbf{R}_3$.

This show a certain *lack of modularity which we want to avoid.* Therefore it shall be possible to write the different agents independently of each other only knowing the interface to $\mathbf{A}$, i.e. $\mathbf{R}_i$ knows λ of $\mathbf{A}$ and $\mathbf{Q}$ knows γ of $\mathbf{A}$.

As a consequence it must be possible for $\mathbf{R}_i$ to have a label which in a certain sense is unique in all contexts and which later can be used as communication link between $\mathbf{R}_i$ and $\mathbf{Q}$. In addition the label shall remain unique or private to $\mathbf{R}_i$ and $\mathbf{Q}$ after it is communicated through $\mathbf{A}$ via λ and γ, (except of course if it is communicated further from either $\mathbf{Q}$ or $\mathbf{R}_i$).

Last but not least we impose the restriction to keep as close as possible to Milners CCS - e.g. *preserving as many as possible of the algebraic properties of CCS.* This requirement is actually quite independent of the extension we are after. With all the thoughts behind and the elegancy of CCS this must be a sound principle to apply to any attempt at extending CCS. We will now give an idea of how ECCS can be made in order to meet these requirements.

Milner has already drawn attention to the connection between the binder "?" in $\alpha?x.$— of CCS and the binder "λ" in the lambda-calculus ([Mil1, p.49]). He has also introduced a textual substitution postfix which has similar characteristics as the substitution prefix of the lambda-calculus (see [Mil1, p.67]) namely: when applied change of bound variables is done as necessary to avoid clashes. It is clear that the substitution postfix formally can be handled along the lines of the substitution prefix of the lambda-calculus as long as we only are concerning the binding construct $\alpha?x.$—. But it is less clear that the binding construct fix $\tilde{X}$ introduced for recursion in [Mil2, Mil3] can be handled formally within the same framework, especially when $\tilde{X} = \langle X_i; i \in I\rangle$ an I-indexed family of distinct variables where I is a uncountable set. One of our aims will therefore be to lift the results for the extension where fix appears as binder too (only for finite $\tilde{X}$).

In order to meet the requirement of independence of actual names used for restriction we will furthermore widen the idea of bound and free occurrences to include labels as well, with "$\backslash$" as the binding symbol for labels. In the lambda-calculus a central notion is α-convertibility between functions with respect to bound variables. The idea is that functions which are equal "up to bound variables" denotes the same function when applied to the same arguments. With our requirement that behaviour expressions which are equal "up to bound labels" should behave equal, it seems natural to extend the notion of α-convertibility to include labels bound by "$\backslash$".

The close relationship between substitution of variables and α-convertibility (in the following just convertibility) will therefore also be generalized to substitution of labels. At the same time we thereby obtain the possibility to change unbound labels of a behaviour expression (i.e. the sort) and can therefore omit the postfixed relabelling operator. For the third requirement (possibility to have a unique or private label, to communicate it and remain unique) notice that the label α in some sense is unique to B in $B \setminus \alpha$ since α can be used for internal communication and cannot interfere with other

α's appearing in any contexts $B \setminus \alpha$ could be in. So in order to communicate α it must be possible to extend α's scope to include the recipient. This and the remaining uniqueness is obtained through a (minor) extension of the inference rules.

To put the comments above differently, we want to extend CCS to allow passing of individual channels, viewing restriction as a formal binder, and to allow dynamic change of scope of such binders in connection with communication.

In order to get an idea of the possibilities of ECCS we turn back to the example of an administrator **A** and some agents $\mathbf{Q}_i$ requesting for some resources $\mathbf{R}_j$ via the administrator. We will make the simplifying assumption that it does not matter what resource a requesting agent gets, though the resources may be implemented differently as long as they obey the same communication protocol. If the administrator use the “first come first served” policy it can be implemented as a FIFO-queue where a requesting agent enters the queue at the rear and leaves it at the front when a resource is available:

$$\begin{aligned} \mathbf{A} &= (\gamma?.\textit{newreq}.\mathbf{T}(\textit{newreq},\sigma) \mid \sigma!.\mathrm{nil}) \setminus \sigma, \text{where} \\ \mathbf{T} &= \mathrm{fix}\, X\langle \textit{oldreq}, \textit{x}\rangle\langle(\gamma?\textit{newreq}.X(\textit{newreq},\sigma) \mid \\ &\qquad\qquad \textit{x}?.\lambda?\textit{freeres}.\textit{oldreq}!\textit{freeres}.\sigma!.\mathrm{nil}) \setminus \sigma\rangle \end{aligned}$$

Each slanted name denotes a label variable. $(\lambda, \gamma \neq \sigma)$.

The administrator can be viewed as consisting of a series of elements each containing a waiting agent (*oldreq*). At the rear new request are received ($\gamma?$*newreq*) and a new element created. The front element receives the name of a available resource ($\lambda?$*freeres*) and passes it on to the waiting agent (*oldreq*!*freeres*). After doing this it signals to the next ($\sigma!$) that it now is the new front element. Notice the elements at the front and rear can serve the agents and resources concurrently. We now turn to the parts of the resources and requesting agents which concerns the communication between them and the administrator. A resource could look something like:

$$\mathbf{R}_j = \mathrm{fix}\, X\tilde{p}\langle(\lambda!\alpha.\underline{\alpha?x}.\ldots\underline{\alpha!e}.X(\ldots)) \setminus \alpha\rangle, \alpha \neq \lambda$$

and a requesting agent:

$$\mathbf{Q}_i = \ldots(\gamma!\beta.\beta?\textit{x}.\underline{x!e'}.\ldots\underline{x?y}.\ldots) \setminus \beta\ldots, \beta \neq \gamma$$

The underlined actions corresponds to the communication protocol between requesting agents and resources for this special example. The other shown actions concerns the communication with the administrator. The label α in the resource is restricted and therefore unique or private for $\mathbf{R}_j$. Upon sending it to the administrator it's scope is extended to include the administrator (in accordance with Com$\rightarrow$(3) and Res$\rightarrow$(2) in section 4). A new name is possibly chosen in order not to interfere with other labels within the new scope such that it remains private to the resource. Afterwards it is passed on to the requesting agent at the front of the queue and the scope include the agent in the same manner as before. The same happens when the requesting agent sends it's private label — which acts like a identification — to the administrator. Notice that a resource after it has served an agent restores itself such that it sends a new private label when reporting that it is available.

To clarify the idea let us consider a possible development of a very simple scenery with two agents and one resource:

$$\mathbf{Q}_1 = (\gamma!\beta.\beta?\mathsf{x}.\mathsf{x}!8.\mathsf{x}?y.\delta!y.\text{nil}) \setminus \beta$$
$$\mathbf{Q}_2 = (\gamma!\beta.\beta?\mathsf{x}.\mathsf{x}!5.\mathsf{x}?y.\varepsilon!y.\text{nil}) \setminus \beta$$
$$\mathbf{R} = \text{fix}\, X\langle(\lambda!\alpha.\alpha?x.\alpha!2 * x.X) \setminus \alpha\rangle$$

The system could be set up as:

$$(\mathbf{Q}_1 \mid \mathbf{Q}_2 \mid \mathbf{A} \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

If the derivations and equivalences in the following possible development is not clear to the reader we refer to section 4 and 5.

$$(\mathbf{Q}_1 \mid \mathbf{Q}_2 \mid \mathbf{A} \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

$$\big\downarrow \tau$$

$$((\overbrace{\beta'?x.x!8.x?y.\delta!y.\mathrm{nil}}^{\mathbf{Q}_1'} \mid \mathbf{Q}_2 \mid (\mathbf{T}(\beta',\sigma') \mid \sigma'!.\mathrm{nil}) \setminus \sigma') \setminus \beta' \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

$$\downarrow \tau$$

$$((\mathbf{Q}_1' \mid \mathbf{Q}_2 \mid ((\gamma?\mathit{newreq}.\mathbf{T}(\mathit{newreq},\sigma'') \mid \lambda?\mathit{freeres}.\beta'!\mathit{freeres}.\sigma''!.\mathrm{nil}) \setminus \sigma'' \mid \mathrm{nil}) \setminus \sigma') \setminus \beta' \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

$$\sim$$

$$((\mathbf{Q}_1' \mid \mathbf{Q}_2 \mid (\gamma?\mathit{newreq}.\mathbf{T}(\mathit{newreq},\sigma'') \mid \lambda?\mathit{freeres}.\beta'!\mathit{freeres}.\sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta' \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

$$\big\downarrow \tau$$

$$((\mathbf{Q}_1' \mid (\overbrace{\beta''?x.x!5.x?y.\varepsilon!y.\mathrm{nil}}^{Q_2'} \mid (\mathbf{T}(\beta'',\sigma'') \mid \lambda?\mathit{freeres}.\beta'!\mathit{freeres}.\sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta'') \setminus \beta' \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

$$\downarrow \tau$$

$$((\mathbf{Q}_1' \mid (\mathbf{Q}_2' \mid (\mathbf{T}(\beta'',\sigma'') \mid \beta'!\alpha'.\sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta'') \setminus \beta' \mid \alpha'?x.\alpha'!2*x.\mathbf{R}) \setminus \alpha' \setminus \gamma \setminus \lambda$$

$$\downarrow \tau$$

$$((\alpha'!8.\alpha'?y.\delta!y.\mathrm{nil} \mid (\mathbf{Q}_2' \mid (\mathbf{T}(\beta'',\sigma'') \mid \sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta'') \setminus \beta' \mid \alpha'?x.\alpha'!2*x.\mathbf{R}) \setminus \alpha' \setminus \gamma \setminus \lambda$$

$$\sim$$

$$(\alpha'!8.\alpha'?y.\delta!y.\mathrm{nil} \mid (\mathbf{Q}_2' \mid (\mathbf{T}(\beta'',\sigma'') \mid \sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta'' \mid \alpha'?x.\alpha'!2*x.\mathbf{R}) \setminus \alpha' \setminus \gamma \setminus \lambda$$

$$\downarrow \tau$$

$$(\alpha'?y.\delta!y.\mathrm{nil} \mid (\mathbf{Q}_2' \mid (\mathbf{T}(\beta'',\sigma'') \mid \sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta'' \mid \alpha!16.\mathbf{R}) \setminus \alpha' \setminus \gamma \setminus \lambda$$

$$\downarrow \tau$$

$$(\delta!16.\mathrm{nil} \mid (\mathbf{Q}_2' \mid (\mathbf{T}(\beta'',\sigma'') \mid \sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta'' \mid \mathbf{R}) \setminus \alpha' \setminus \gamma \setminus \lambda$$

$$\sim$$

$$(\delta!16.\mathrm{nil} \mid (\mathbf{Q}_2' \mid (\mathbf{T}(\beta'',\sigma'') \mid \sigma''!.\mathrm{nil}) \setminus \sigma'') \setminus \beta'' \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

$$\vdots$$

$$(\delta!16.\mathrm{nil} \mid \varepsilon!10.\mathrm{nil} \mid \mathbf{A} \mid \mathbf{R}) \setminus \gamma \setminus \lambda$$

It should be stressed that following the guidelines indicated above one obtain the required modularity mentioned earlier. The administrator can be constructed without knowing anything about the resources and the requesting agents except that the agents makes requests via γ and resources reports their availability at λ. Similar for new resources and requesting agents one whish to add to the system. They can be added at any time without any inspection of the components already in the system. For instance we could add a new resource:

$$\mathbf{N} = \mathrm{fix}\, X\langle(\lambda!\varepsilon.\varepsilon?x.\alpha!x + x.X) \setminus \varepsilon\rangle$$

directly and the system would look like:

$$(\mathbf{Q}_1 \mid \mathbf{Q}_2 \mid \mathbf{A} \mid \mathbf{R} \mid \mathbf{N}) \setminus \gamma \setminus \lambda$$

No confusion of identifications etc. can arise. This way of adding new components to the system seems lucid compared to the "index-approach" where one have to inspect the system in detail.

We will now present ECCS in detail.

2 Syntax

The intension of this section is to give a detailed presentation of the syntax of ECCS and to state conventions and notation. For most purposes it will do to look at the syntax table and the section should be used for reference when doubt arise.

In the original CCS details of types of variables and value expressions are avoided in favour to a more clear presentation. When allowing label passing we cannot avoid all such considerations. Basicly we distinguish three types of values:

- recursion definitions
- labels
- and other values.

The last two can be communicated. Of course the set of variables of a certain type is *disjoint* to the corresponding set of values. Similar we assume sets of different types to be disjoint. We display explicit four such disjoint sets:

Value variables:	$\bar{x}, \bar{x}_0, \ldots, \bar{y}, \bar{y}_0, \ldots$
Label variables:	$\mathsf{x}, \mathsf{x}_0, \ldots, \mathsf{y}, \mathsf{y}_0, \ldots$
Recursion variables:	$X, X_0, \ldots, Y, Y_0, \ldots$
Labels (or names):	$\alpha, \alpha_0, \ldots, \beta, \beta_0, \ldots$ (the set denoted Δ)

which we assume are ordered and never exhausted. We also have the symbol τ which does not belong to any sets, especially $\tau \notin \Delta$. If nothing else mentioned the following symbols will denote:

a: label or label variable,

b, c: variable or label (bindable elements)

e: label, label variable or a value expression built from constant and function symbols as usual (e.g. used as actual parameters to recursion definitions),

f: value expression alone,

$p, x, x_0, \ldots, y, y_0, \ldots$: label variables or value variables (e.g. used for formal parameters to recursion definitions),

v: label or value,

M, N: label, label variable,value expression, recursion variable or a indexed recursion definition (see later),

$B, B_0, \ldots, B', B'', \ldots E, E_0, \ldots$: behaviour expressions.

(Obviously we don't expect the reader to memorise these notations, but they do ease the presentation of our material, and the reader may inspect this list in case of doubt.)

We are actually considering labels as values too (v denote also a label) but do only make an explicit distinction between different types of values: labels and "other" values.

From the ordering of the different sets lists are formed:

List$_b$ is defined to be the list obtained by ordering the set to which b belongs.

So if the labels are ordered as indicated above then e.g.

$$\text{List}_\gamma = \alpha, \alpha_0, \ldots, \beta, \beta_0, \ldots$$

Lists are used in the definition of the substitution prefix in the next section in order to make it unambiguous.

Vectors are used quite often and will be abbreviated. For instance the vector $\langle X_1, \ldots, X_n \rangle$ will be abbreviated $\tilde{X}$. If the dimension of the vector is of interest it will be indicated by placing it below to the right, i.e. in the example $\tilde{X}_n$ indicates that $\tilde{X}$ has dimension n.

In most cases when dealing with vectors of variables they shall meet the requirement that

all variables in a vector of variables are mutual different.

Therefore we will assume this requirement to be fulfilled except when something else mentioned.

If a function is defined on the elements of a set resulting in a subset of another set it is extended in the natural way to include vectors of these elements too. For instance if FV is the function giving the free variables of a behaviour expression then

$$\mathrm{FV}(\tilde{E}) = \bigcup_{i\in\{1,\ldots,n\}} \mathrm{FV}(E_i) \text{ and } \mathrm{FV}(\tilde{E}_0) = \mathrm{FV}(\langle\rangle) = \emptyset$$

We will define four such functions described intuitively by

Free variables and labels:

FV(B) the free variables occuring in B, e.g. $y \in \mathrm{FV}(\lambda?x.\lambda!x + y.\ldots)$.

L(B) the free labels in B, e.g. $\lambda \in \mathrm{L}(\lambda?x.\lambda!x + y.\ldots)$. I.e. L($B$) is the sort of B.

$\mathcal{F}(B) := \mathrm{FV}(B) \cup \mathrm{L}(B)$.

Bound variables and labels:

$\mathcal{B}(B)$ is the bound labels and variables, e.g. $x, \alpha \in \mathcal{B}((\lambda?x.\lambda!x + y.\ldots) \setminus \alpha)$.

The function FV applied to a vector of variables will be abbreviated by placing curly brackets around it, e.g. FV($\tilde{X}$) is written $\{\tilde{X}\}$. Similar for expressions: $\mathrm{FV}(\tilde{e}) = \{\tilde{e}\}$.

Two vectors are said to be *comparable* if they have the same dimension and corresponding elements are of the same type, i.e.:

$$\tilde{M}_k \text{ comparable to } \tilde{N}_n \text{ iff } k = n \text{ and for each } i\ (1 \le i \le k)\ M_i \text{ is of same type as } N_i.$$

We say that two *vectors are equal* if they are comparable and the corresponding elements are equal:

$$\tilde{M} = \tilde{N} \text{ iff } M_i = N_i \text{ for all } i$$

If two vectors $\tilde{M}$ and $\tilde{N}$ have *no free occuring elements in common* we write $\tilde{M} \sharp \tilde{N}$:

$$\tilde{M} \sharp \tilde{N} \text{ iff } \mathcal{F}(\tilde{M}) \cap \mathcal{F}(\tilde{N}) = \emptyset.$$

Form	B''	$\mathrm{FV}(B'')$	$\mathrm{L}(B'')$	$\mathcal{B}(B'')$
Inaction	$\underline{\mathrm{nil}}$	$\emptyset$	$\emptyset$	$\emptyset$
Action	$\lambda\underline{?}y\underline{.}B$	$\mathrm{FV}(B) - \{y\}$	$\mathrm{L}(B) \cup \{\lambda\}$	$\mathcal{B}(B) \cup \{y\}$
	$\mathsf{x}\underline{?}y\underline{.}B$	$(\mathrm{FV}(B) - \{y\}) \cup \{\mathsf{x}\}$	$\mathrm{L}(B)$	$\mathcal{B}(B) \cup \{y\}$
	$\lambda\underline{!}e\underline{.}B$	$\mathrm{FV}(B) \cup \{e\}$	$\mathrm{L}(B) \cup \mathrm{L}(e)$	$\mathcal{B}(B) \cup \mathcal{B}(e)$
	$\mathsf{x}\underline{!}e\underline{.}B$	$\mathrm{FV}(B) \cup \{e\} \cup \{\mathsf{x}\}$	$\mathrm{L}(B) \cup \mathrm{L}(e)$	$\mathcal{B}(B) \cup \mathcal{B}(e)$
	$\tau\underline{.}B$	$\mathrm{FV}(B)$	$\mathrm{L}(B)$	$\mathcal{B}(B)$
Summation	$B\underline{+}B'$	$\mathrm{FV}(B) \cup \mathrm{FV}(B')$	$\mathrm{L}(B) \cup \mathrm{L}(B')$	$\mathcal{B}(B) \cup \mathcal{B}(B')$
Composition	$B\underline{\|}B'$	$\mathrm{FV}(B) \cup \mathrm{FV}(B')$	$\mathrm{L}(B) \cup \mathrm{L}(B')$	$\mathcal{B}(B) \cup \mathcal{B}(B')$
Restriction	$B\underline{\backslash}\alpha$	$\mathrm{FV}(B)$	$\mathrm{L}(B) - \{\alpha\}$	$\mathcal{B}(B) \cup \{\alpha\}$
Recursion	$\underline{\mathrm{fix}}_i\tilde{X}\tilde{p}\tilde{E}\underline{(}\tilde{e}\underline{)}$	$(\mathrm{FV}(\tilde{E}) - \{\tilde{X},\tilde{p}\}) \cup \{\tilde{e}\}$	$\mathrm{L}(\tilde{E}) \cup \mathrm{L}(\tilde{e})$	$\mathcal{B}(\tilde{E}) \cup \{\tilde{X},\tilde{p}\} \cup \mathcal{B}(\tilde{e})$
	$X\underline{(}\tilde{e}\underline{)}$	$\{X\} \cup \{\tilde{e}\}$	$\mathrm{L}(\tilde{e})$	$\mathcal{B}(\tilde{e})$
Conditional	$\underline{\textbf{if}}\ f\ \underline{\textbf{then}}\ B$ $\underline{\textbf{else}}\ B'$	$\{f\} \cup \mathrm{FV}(B) \cup \mathrm{FV}(B')$	$\mathrm{L}(B) \cup \mathrm{L}(B')$	$\mathcal{B}(f) \cup \mathcal{B}(B) \cup \mathcal{B}(B')$

Table 1: Syntax table for behaviour expressions.

E.g. for vectors of variables: $\tilde{x} \sharp \tilde{x}'$ iff $\{\tilde{x}\} \cap \{\tilde{x}'\} = \emptyset$.

Let A be a set. Then we write

$$\tilde{M} \notin A \text{ iff } \mathcal{F}(\tilde{M}) \cap A = \emptyset$$

In what comes it will be useful with some terminology about recursion. A *recursion definition* is $\mathrm{fix}\,\tilde{X}\tilde{p}\tilde{E}$. A recursion definition can be indexed as $\mathrm{fix}_i\,\tilde{X}\tilde{p}\tilde{E}$. $\tilde{p}$ is called the *formal parameters* and in $\mathrm{fix}_i\,\tilde{X}\tilde{p}\tilde{E}(\tilde{e})$ or $X(\tilde{e})$, $\tilde{e}$ is called the *actual parameters*. Intuitively $\mathrm{fix}\,\tilde{X}\tilde{p}\tilde{E}(\tilde{v})$ is the solution to the equations: $\tilde{X} = \tilde{E}'$, where $\tilde{E}'$ is the result of substituting $\tilde{v}$ for $\tilde{p}$ in $\tilde{E}$. A recursion variable X_i is comparable to $\mathrm{fix}_i\,\tilde{X}\tilde{p}\tilde{E}$ — an indexed recursion definition.

The fixed elements of the syntax is underlined. We adopt the usual constructor precedence of CCS, i.e.

$$\text{Restriction} > \text{Action} > \text{Composition} > \text{Summation} > \text{Conditional}$$

If $\tilde{e}$ contains no elements we write X instead of $X(\langle\rangle)$ and similar if $\tilde{p}$ is a empty vector we write $\mathrm{fix}_i\,\tilde{X}\tilde{E}$ in stead of $\mathrm{fix}_i\,\tilde{X}\langle\rangle\tilde{E}()$. Also if $\tilde{X} = \tilde{X}_1$ we write $\mathrm{fix}\,X\tilde{p}E(\tilde{e})$ in stead of $\mathrm{fix}_1\langle X\rangle\tilde{p}\langle E\rangle(\tilde{e})$.

Notes to the syntax table

i) The x in front of $\underline{?}$ is a label variable which is supposed to be bound to a label.

ii) Notice that labels and label variables also are allowed as actual parameters to a recursion expression.

iii) In a conditional expression no label or label variable may be contained in f to form a boolean expression.

Constraints on recursion expressions

In recursion definitions as $\mathrm{fix}\tilde{X}\tilde{p}\tilde{E}$, $\tilde{X}$ and $\tilde{E}$ shall be comparable. If $\tilde{X}$ has dimension n then $\mathrm{fix}\tilde{X}\tilde{p}\tilde{E}$ may only be indexed by $i \in \{1, \ldots, n\}$ (except as mentioned when $n = 1$). When a indexed recursion definition appears on the form $\mathrm{fix}_i\tilde{X}\tilde{p}\tilde{E}(\tilde{e})$ then $\tilde{p}$ and $\tilde{e}$ must be comparable. A recursion variable X can occur free in several places in a behaviour expression. Let $X(\tilde{e})$ and $X(\tilde{e}')$ be two such occurrences. Then $\tilde{e}$ and $\tilde{e}$' must be comparable. E.g. the following example would make no sense:

$$\alpha?y.X(y) + \beta?y.\gamma?z.X(y, z)$$

We will state a further constraint on recursion expressions namely that they are *guardedly well-defined.* Though it can be checked at a syntactical level we will postpone it to the section of derivations because it is motivated through it's consequences for the derivations of a behaviour expression.

As perhaps indicated by the syntax table above behaviour expressions may only be finite, i.e. a behaviour expression shall consist of finitely many subterms. The constraint of guardedly well-definedness has as one of it's consequences that all behaviour expressions derived by finitely many actions from a certain behaviour expression also will be finite.

3 Substitution, Conversion

On the following pages we are going to define single substitution, i.e. a expression substituted for *one* element, e.g. $[M/b]$. The definition will make use of multiple substitution (or multiple replacement). Multiple substitution can be defined in terms of single substitution:

Definition 1 *Let $\tilde{M}$ and $\tilde{b}$ be comparable and assume that either $\tilde{b} \notin \mathcal{F}(\tilde{M})$ or $\tilde{b} = \tilde{M}$.*

Then the multiple substitution *of $\tilde{M}$ for $\tilde{b}$ in B: $[\tilde{M}/\tilde{b}]B$ is defined recursively as follows ($\tilde{M} = \langle M_1, \ldots, M_n\rangle, \tilde{b} = \langle b_1, \ldots, b_n\rangle$):*

$$\begin{array}{ll} [\tilde{M}/\tilde{b}]B = [M_1/b_1]B & \textit{if } n = 1 \\ [\tilde{M}/\tilde{b}]B = [M_n/b_n][\tilde{M}_{n-1}/\tilde{b}_{n-1}]B & \textit{if } n > 1 \end{array} \qquad \square$$

As an example let $\tilde{M} = \text{fix}\,\tilde{X}\tilde{E} = \langle \text{fix}_1\,\tilde{X}\tilde{E},\ \text{fix}_2\,\tilde{X}\tilde{E}\rangle$, $\tilde{b} = \langle X_1, X_2\rangle$. Then $[\text{fix}\,\tilde{X}\tilde{E}/\tilde{X}]B$ is really just a shorthand for $[\text{fix}_2\,\tilde{X}\tilde{E}/X_2][\text{fix}_1\,\tilde{X}\tilde{E}/X_1]B$. When we write $[\tilde{M}/\tilde{b}]$ in the following we will assume the conditions of definition 1 to be satisfied. It should be clear that there cannot be any interference between free variables of $\tilde{b}$ and $\tilde{M}$, see also [BAR, p.30]. Notice that our definition is not a general definition of multiple substitution. We have not found it necessary to use a general definition for our purpose and it would only complicate the proofs. A general definition can be found in [CFC, p.103].

The substitution prefix can be extended to vectors by the following definition:

Definition 2 $[M/b]\tilde{N}_n$ *is defined to be the* vector $\langle [M/b]N_1, \ldots, [M/b]N_n\rangle$. □

We are now ready to give the formal definition of single substitution prefix:

Definition 3 (generalized substitution prefix) *Let M and b be comparable elements as described on page 11($k = 1$). Then $[M/b]B$ is $B*$ defined as follows:*

Case 1) B a label, variable or constant.

(a) If $B = b$ then $B = M$* *(b) If $B \neq b$ then $B* = B$*

Case 2) B an ordinary construction.

(a) If $B = \text{nil}$ then $B = B$* *(b) If $B = a!e.C$ then $B* = a*!.e*.C*$*

(c) If $B = \tau.C$ then $B* = \tau.B*$ *(d) If $B = C + D$ then $B* = C* + D*$*

(e) If $B = C \,|\, D$ then $B* = C* \,|\, D*$(f) If $B = X(\tilde{e})$ then $B* = X*(\tilde{e}*)$

*(g) If $B =$ **if** f **then** C **else** D, then $B* =$ **if** $f*$ **then** $C*$ **else** $D*$*

Case 3) B a binding construct.

I: $B = a?x.C$

(a) If $b = x$ then $B = a*?x.C$*

(b) If $b \neq x$ then $b = a*?[M/b][y/x]C$, where y is the variable defined as:*

i) If $b \notin \mathcal{F}(C)$ or $x \notin \text{FV}(M)$ then $y = x$

ii) If $b \in \mathcal{F}(C)$ and $x \in \text{FV}(M)$ then $y \neq b$ is the first *variable in* List_x

such that $y \notin \text{FV}(C) \cup \text{FV}(M)$

II: $B = C \setminus \alpha$

(a) If $b = \alpha$ then $B = B$*

(b) If $b \neq \alpha$ then $B* = [M/b][\beta/\alpha]C \setminus \beta$, where β is the label defined as:

i) If $b \notin \mathcal{F}(C)$ *or* $\alpha \notin \mathrm{L}(M)$ *then* $\beta = \alpha$

ii) If $b \in \mathcal{F}(C)$ *and* $\alpha \in \mathrm{L}(M)$ *then* $\beta \neq b$ *is the* first *label in* List_{α}

such that $\beta \notin \mathrm{L}(C) \cup \mathrm{L}(M)$

III: $B = \mathrm{fix}_i\, \tilde{X}\tilde{p}\tilde{E}(\tilde{e})$

(a) If $b \in \{\tilde{X}, \tilde{p}\}$ *then* $B* = \mathrm{fix}_i\, \tilde{X}\tilde{p}\tilde{E}(\tilde{e}*)$

(b) If $b \notin \{\tilde{X}, \tilde{p}\}$ *then* $B* = \mathrm{fix}_i\, \tilde{X}'\tilde{p}'[M/b][\tilde{X}'\tilde{p}'/\tilde{X}\tilde{p}]\tilde{E}(\tilde{e}*)$, *where* $\tilde{X}'$ *and* $\tilde{p}'$ *are defined as:*

i) If $b \notin \mathcal{F}(\tilde{E})$ *or* $\tilde{X}$, $\tilde{p} \notin \mathrm{FV}(M)$ *then* $\tilde{X}'$, $\tilde{p}' = \tilde{X}$, $\tilde{p}$

ii) If $b \in \mathcal{F}(\tilde{E})$ *and* $\{\tilde{X}, \tilde{p}\} \cap \mathrm{FV}(M) \neq \emptyset$ *then* $\tilde{X}'$, $\tilde{p}'$ *are chosen*
$b \notin \{\tilde{X}', \tilde{p}'\}, \tilde{X}', \tilde{p}' \sharp \tilde{X}, \tilde{p}$ *and*
if $\tilde{X} = \tilde{X}_n$ *then for* $1 \leq k \leq n : X'_k$ *is the* first *variable in* List_{X_k}

such that $\tilde{X}'_k \notin \mathrm{FV}(\tilde{E}) \cup \mathrm{FV}(M) \cup \{\tilde{X}'_{k-1}\}$, *and*
if $\tilde{p} = \tilde{p}_n$ *then for* $1 \leq k \leq n : p'_k$ *is the* first *variable in* List_{p_k}

such that $\tilde{p}'_k \notin \mathrm{FV}(\tilde{E}) \cup \mathrm{FV}(M) \cup \{\tilde{p}'_{k-1}\}$

The definition is extended to include substitutions in indexed recursion definitions by means of case 3.III: If $[M/b]\, \mathrm{fix}_i\, \tilde{X}\tilde{p}\tilde{E}(\tilde{e}) = \mathrm{fix}_i\, \tilde{X}'\tilde{p}'\tilde{E}'(\tilde{e})$ *then* $[M/b]\, \mathrm{fix}_i\, \tilde{X}\tilde{p}\tilde{E} = \mathrm{fix}_i\, \tilde{X}'\tilde{p}'\tilde{E}'$. □

There are several notes to make about this definition:

i) To avoid to much use of parentheses we assume the substitution prefix to take precedence over the constructors of CCS. So $[M/b]B \setminus \alpha$ means $([M/b]B) \setminus \alpha$.

ii) A lack in the definition is seen in case 2.b,f,g and case 3 where the substitution prefix is used on expressions: e, f. If e is a variable, a label or another constant the situation is handled by case 1. But for other value expressions there is no explicit definition. We will assume all such expressions are written in the notation of the lambda-calculus and do extend the definition (implicit) with the definition from there. The same will be with the convertibility relation we are going to define. The theory for conversion from the lambda-calculus can therefore be lifted to these subparts of a behaviour expression and will not be considered further in the proofs to come.

iii) The definition uses in case 3.I-III.b implicitly the notion of independence of actual names of labels and variables. This notion of independence is captured in the conversion concept which is presented formally hereafter.

iv) The use of choosing the *first* label or variable in a list fulfilling a certain condition in case 3.I-III.b.ii makes the substitution unambiguous.

v) Although the symbol B normal wise is used for behaviour expressions alone it is here used for atomic elements too as seen in case 1. This is done to simplify the definition. So the substitution prefix is defined not only for behaviour expressions but also for atomic elements as labels, values and variables. E.g. $[\beta/\alpha]5 = 5$.

The notion of a expression context will be used in defining which behaviour expressions there are convertible. Intuitively a expression context is a behaviour expression with some holes in it where another behaviour expression can be placed. Formally:

Definition 4 *An* expression context *(just context for short) — written* $\mathcal{C}[]$ *— is defined as follows:*

i) $[\,]$ *is a context.* B *— a behaviour expression — is a context*

If $\mathcal{C}[\,]$ *is a context then so are* $a?x.\mathcal{C}[\,]$, $a!e.\mathcal{C}[\,]$, $\tau.\mathcal{C}[\,]$ *and* $\mathcal{C}[\,]\setminus\alpha$

If $\mathcal{C}[\,]$ *and* $\mathcal{C}'[\,]$ *are contexts then the following is a context* $\mathcal{C}[\,]+\mathcal{C}'[\,]$, $\mathcal{C}[\,]\mid\mathcal{C}'[\,]$, *and* **if** f **then** $\mathcal{C}[\,]$ **else** $\mathcal{C}'[\,]$

If $\mathcal{C}_i[\,](1 \leq i \leq n)$ *is a context then so is* $\mathrm{fix}_i\,\tilde{X}\tilde{p}\langle\mathcal{C}_1[\,],\ldots,\mathcal{C}_n[\,]\rangle(\tilde{e})$.

ii) If $\mathcal{C}[\,]$ *is a context and* B *a behaviour expression then* $\mathcal{C}[B]$ *denotes the result of replacing all* $[\,]$ *in* $\mathcal{C}[\,]$ *(if any) by* B.

$\square$

Note to ii): If $\mathcal{C}[\,]$ contains no $[\,]$'s then $\mathcal{C}[B] = \mathcal{C}[\,]$. In general free labels or variables of B may become bound. For instance if $B = \lambda!5.\mathrm{nil}$ and $\mathcal{C}[\,] = (\lambda?x.\mathrm{nil}\mid[\,])\setminus\lambda$ then $\mathcal{C}[B] = (\lambda?x.\mathrm{nil}\mid B = \lambda!5.\mathrm{nil})\setminus\lambda$ and λ of B is bound.

We are now able to introduce the convertibility relation among behaviour expressions. The name of the different rules are chosen from [CFC] and [BAR] in which they are used in the lambda-calculus for similar rules.

Definition 5 (of cnv, convertible) cnv *is a substitutive equivalence relation:*

(ϱ) $B \ \mathsf{cnv}\ B$ *(reflexive)*

(σ) $\dfrac{B \ \mathsf{cnv}\ C}{C \ \mathsf{cnv}\ B}$ *(symmetric)*

(τ) $\dfrac{B \ \mathsf{cnv}\ C, C \ \mathsf{cnv}\ D}{B \ \mathsf{cnv}\ D}$ *(transitive)*

$(subs)$ $\dfrac{B \ \mathsf{cnv}\ D, \mathcal{C}[\,]\ \textit{a context}}{\mathcal{C}[B] \ \mathsf{cnv}\ \mathcal{C}[D]}$

conversion rule:

(α_1) $a?x.B \ \mathsf{cnv}\ a?y.[y/x]B$, *provided* $y \neq x$ *and* $y \notin \mathrm{FV}(B)$

(α_2) $B \setminus \alpha \ \mathsf{cnv}\ [\beta/\alpha]B \setminus \beta$, *provided* $\beta \neq \alpha$ *and* $\beta \notin \mathrm{L}(B)$

(α_3) $\mathrm{fix}_i \tilde{X}\tilde{p}\tilde{E}(\tilde{e}) \ \mathsf{cnv}\ \mathrm{fix}_i \tilde{X}'\tilde{p}'[\tilde{X}'\tilde{p}'/\tilde{X}\tilde{p}]\tilde{E}(\tilde{e})$,
provided $\tilde{X}', \tilde{p}' \sharp \tilde{X}, \tilde{p}$ *and* $\tilde{X}', \tilde{p}' \notin \mathrm{FV}(\tilde{E})$

cnv *is extended to include recursion definitions:*

$\mathrm{fix}\tilde{X}\tilde{p}\tilde{E} \ \mathsf{cnv}\ \mathrm{fix}\tilde{X}'\tilde{p}'\tilde{E}'$ *iff* $\mathrm{fix}_i \tilde{X}\tilde{p}\tilde{E}(\tilde{e}) \ \mathsf{cnv}\ \mathrm{fix}_i \tilde{X}'\tilde{p}'\tilde{E}'(\tilde{e})$ *for all (possible)* i *and* $\tilde{e}$. □

(α_1)–(α_3) says that change of bound variable may be done as long as no confusion results. (α_2) corresponds directly to α-conversion of the lambda-calculus and (α_1) too except that there is added an element outside the scope of the binder. For theorems of conversion generalized from the lambda-calculus they will therefore be part of proofs in much the same way. Hence the parts of the proofs considering these cases will not always be shown in all detail.

For recursion definitions on the form $\mathrm{fix}_i \tilde{X}\tilde{E}$ where $\tilde{X}$ is of finite dimension (as by us) Hennesey has argued that it is enough to consider definitions where $\tilde{X}$ has dimension one, i.e. $\tilde{X} = \langle X \rangle = X$, since a definition can be expanded by replacing X_j by $\mathrm{fix}\langle X_j \rangle E'_j$ where the same is done for E'_j until all recursion variables are bound by a recursion binder with one variable. This would reduce the proofs for the case (α_3) to one similar to (α_1). But we are dealing with recursion definitions possibly with parameters so we cannot avoid the problem.

We will say that two *comparable vectors* of behaviour expressions are *convertible*:

$$\tilde{E} \ \mathsf{cnv}\ \tilde{E}' \text{ iff } E_i \ \mathsf{cnv}\ E'_i (1 \leq i \leq n, \text{ where } \tilde{E} = \tilde{E}_n).$$

Proposition 6 *If* $\mathrm{fix}\tilde{X}\tilde{p}\tilde{E}$ *and* $\mathrm{fix}\tilde{X}\tilde{p}\tilde{E}'$ *are syntactical correct then*

$$\tilde{E} \ \mathsf{cnv}\ \tilde{E}' \Rightarrow \mathrm{fix}\tilde{X}\tilde{p}\tilde{E} \ \mathsf{cnv}\ \mathrm{fix}\tilde{X}\tilde{p}\tilde{E}'.$$

The rest of this section is devoted to the most important properties of the substitution prefix.

Theorem 7 (syntactical equalities)

(a) $[b/b]B = B$

(b) $b \notin \mathcal{F}(B) \Rightarrow [M/b]B = B$

The following properties assumes $(\mathcal{F}(M) \cup \mathcal{F}(N)) \cap \mathcal{B}(B) = \emptyset$.

(c) $[M/b][N/c]B = [[M/b]N/c][M/b]B$, *provided either*
(c_1) $c \neq b$, $c \notin \mathcal{F}(M)$ *or*
(c_2) $b \notin \mathcal{F}(B)$

(d) $[M/b][N/b]B = [[M/b]N/b]B$

Proof Based on induction on the *rank* of B — the "syntactical depth" of B — using the classification into cases as in definition 3. We omit the formal definition of rank (**Definition 8**) as it is mainly of proof technical interest. In (c) and (d) the result of theorem 9 below is used. □

Notice that if (c_2) holds then (c) by (b) reads $[M/b][N/c]B = [[M/b]N/c]B$.

The following theorem characterize the free bindable elements of an expression with substitution.

Theorem 9

$$c \in \mathcal{F}([M/b]B) \text{ iff either } \begin{array}{l} c \neq b, c \in \mathcal{F}(B) \text{ or} \\ b \in \mathcal{F}(B), c \in \mathcal{F}(M) \end{array}$$

Proof Induction on the rank of B — using (a) and (b) of theorem 7. □

Intuitively theorem 9 says that c occurs free in $[M/b]B$ iff either it already occured free in B before the substitution or it occurs free in M which is substituted somewhere in B. Notice that we from theorem 9 get

$$c \notin \mathcal{F}(B) \cup \mathcal{F}(M) \Rightarrow c \notin \mathcal{F}([M/b]B).$$

This property is used often, also in later sections.

The following theorem together with theorem 7 are probably some of the most important of the theory of substitution and conversion.

Theorem 10 (conversion equalities)

(a) $B \;\mathsf{cnv}\; C \Rightarrow [M/b]B \;\mathsf{cnv}\; [M/b]C$

(b) $[M/b][N/c]B \;\mathsf{cnv}\; [[M/b]N/c][M/b]B$, *provided either*
(b_1) $c \neq b, c \notin \mathcal{F}(M)$ *or*
(b_2) $b \notin \mathcal{F}(B)$

(c) $[M/b][N/b]B \;\mathsf{cnv}\; [[M/b]N/b]B$

Proof Essential induction on the definition of convertibility from definition 5 (including the structure of $\mathcal{C}[\,]$ for the (subs)-case), and the rank of B and C. □

Intuitively two behaviour expressions are convertible if they have the same free elements and the same structure, differing only in their bound labels and variables. The following theorem states formally the first property. That they have the same structure is immediate from definition 5.

Theorem 11

$$B \;\mathsf{cnv}\; C \Rightarrow \mathcal{F}(B) = \mathcal{F}(C)$$

Proof Essential induction on the definition of convertibility. □

It is natural to expect a pendant to theorem 10 where the condition of convertibility concerns the arguments of the substitution instead and it turns out that the following theorem is true.

Theorem 12

$$M \;\mathsf{cnv}\; N \Rightarrow [M/b]B = [N/b]B$$

Proof Only nontrivial if b is a recursion variable and M, N recursion definitions — proved by induction on the rank of B. □

4 Derivations

As in CCS we also define a binary relation $\xrightarrow{\Gamma}$ (an atomic action relation) over behaviour expressions, but it differs with respect to receiving a value or label. We let the relation qualify over all the values which could be received by writing for instance $\xrightarrow{\lambda\Gamma x}$. Intuitively $B \xrightarrow{\lambda\Gamma x} B'$ means that B become $[v/x]B'$ under $\lambda?v$ where v is any value x can assume, i.e. v is a value comparable to x. The other relations are $\xrightarrow{\lambda!v}$ and $\xrightarrow{\tau}$. $B \xrightarrow{\lambda!v} B'$ means that B becomes B' under $\xrightarrow{\lambda!v}$, i.e. by sending v via λ. For τ ($\tau \notin \Delta$) the relation corresponds to an internal action. We will actually only define $B \xrightarrow{\Gamma} B'$ where B is a certain kind of behaviour expression namely one who has the property that it is a program:

Definition 13 *A* program *is a behaviour expression closed with respect to free variables. I.e.*

$$B \textit{ is a program iff } \mathrm{FV}(B) = \emptyset$$

□

In CCS the atomic action relation is defined over programs. I.e. resulting behaviour expression is also a program, such that the program property is invariant under derivation. This is not the case with ECCS, but a similar property is obtained through the definition of strong and observational equivalence. When looking at the axioms and inference rules below defining the atomic action relation remember note iv) to definition 5 that we only write $[M/b]$ when M and b are comparable. So for instance the inference rules Com→(2) below are only defined when v and x are comparable. Similarly the axiom Act→(2) is only defined for y and x comparable.

Definition 14
Ina→ nil *has no atomic actions*

Act→

(1) $\lambda ?x.B \xrightarrow{\lambda\Gamma y} [y/x]B$

(2) $\lambda !v.B \xrightarrow{\lambda !v} B$

(3) $\tau.B \xrightarrow{\tau} B$

Sum→

L: $\dfrac{B_0 \xrightarrow{\Gamma} B_0'}{B_0 + B_1 \xrightarrow{\Gamma} B_0'}$ *R: Symmetric for* B_1

Com→

(1)L: $\dfrac{B_0 \xrightarrow{\Gamma} B_0'}{B_0 \mid B_1 \xrightarrow{\Gamma} B_0' \mid B_1}$ *R: Symmetric for* B_1

(2)L: $\dfrac{B_0 \xrightarrow{\lambda !v} B_0', B_1 \xrightarrow{\lambda\Gamma x} B_1'}{B_0 \mid B_1 \xrightarrow{\tau} B_0' \mid [v/x]B_1'}$ *R: Symmetric with* B_1 *sending*

(3)L: $\dfrac{B_0 \xrightarrow{\lambda !\mathsf{x}} B_0', B_1 \xrightarrow{\lambda\Gamma\mathsf{x}} B_1', \alpha \notin \mathrm{L}(B_i)}{B_0 \mid B_1 \xrightarrow{\tau} [\alpha/\mathsf{x}](B_0' \mid B_1') \setminus \alpha}$ *R: Sym. exchanging* ! *and* ?

Res→

(1) $\dfrac{B \xrightarrow{\Gamma} B', \alpha \notin \mathrm{L}(?\,)}{B \setminus \alpha \xrightarrow{\Gamma} B' \setminus \alpha}$

(2) $$\frac{B \xrightarrow{\lambda!\alpha} B', \lambda \neq \alpha}{B \setminus \alpha \xrightarrow{\lambda!\mathsf{x}} [\mathsf{x}/\alpha]B'}$$

Rec→

$$\frac{[\mathrm{fix}\,\tilde{X}\tilde{p}\tilde{E}/\tilde{X}][\tilde{v}/\tilde{p}]E_i \xrightarrow{\Gamma} B}{\mathrm{fix}_i\,\tilde{X}\tilde{p}\tilde{E}(\tilde{v}) \xrightarrow{\Gamma} B}$$

Con→

L: $$\frac{B_0 \xrightarrow{\Gamma} B_0'}{\textbf{if } \mathsf{true} \textbf{ then } B_0 \textbf{ else } B_1 \xrightarrow{\Gamma} B_0'}$$ *R: Symmetric for B_1*

□

Before commenting the axioms and inferece rules it is worth to notice:

Proposition 15 *If $B \xrightarrow{\Gamma} B'$ is a part of the inference which ensures $C \xrightarrow{\Gamma} C'$ then $\mathrm{FV}(C) = \emptyset$ implies $\mathrm{FV}(B) = \emptyset$.*

Now for the notes on the action relation:

i) By proposition 15 it is clear that if Act→(1) is the basis of an inference which ensures an action of a program then $\mathrm{FV}(\lambda?x.B) = \emptyset$ and an arbitrary chosen y will therefore be just as good as x. Furthermore it will not interfere with variables in another part of the program since also by proposition 15 any other part of the program which can form an action cannot have free variables. For instance if $D = C \mid \lambda?x.B$ is a program and $D \xrightarrow{\lambda\Gamma x} C \mid B'$ then x cannot be free in C. Therefore it is ensured in Com→(2) that v is substituted in the "right" place. Com→(1), Com→(2) corresponds to those of CCS.

ii) The reason for letting $(\lambda?x.B, [y/x]B)$ be in the the relation $\xrightarrow{\lambda\Gamma y}$ is that we wish convertable programs to be behaviourly equivalent. This could have been obtained through a modification of the definition of strong equivalence, but we have found it more convenient here.

iii) Two inference rules are added to CCS: Com→(3) and Res→(2). They make it possible to extend the scope of a label. Res→(2) cancels the the restriction and for the resons mentioned in i) it does not matter which variable is chosen (as long as it is a label variable of course). In Com→(3) the restriction is placed again and the label is thereby known to ther recipient. The actual name of the label is chosen such that it does not interfer wither other names in the new scope by the condition $\alpha \notin \mathrm{L}(B_i)$. Notice that the original name can be chosen if it does not appear in the enviroment outside the old and inside the new scope.

iv) The definition depends heavily on the substitution prefix and it's properties as seen in Act$\to$(2), Com$\to$(2), Com$\to$(3) and Res$\to$(2). Most of all it depends on the property that a label or variable which "passes" through a bound occurence by substitution changes the names of the bound occurence and "passes" on, thereby avoiding any conflict.

v) The inference rule for the relabelling operation in CCS is missing, but as argued in the section concerning the expressiveness of ECCS the substitution prefix takes over it's role.

The crux of the definition of the atomic action relation is probaly that a variable in some sence can pass via a label. But it has some direct consequences which later will prove useful, e.g.:

Proposition 16 *Assume* $B \overset{\Gamma}{\to} B'$ *(*B *a program). Then*

(a) $\alpha \notin \mathrm{L}(B)$ *implies* $\alpha \notin \mathrm{L}(?) \cup \mathrm{L}(B')$

(b) $\lambda \in \mathrm{L}(?)$ *implies* $\lambda \in \mathrm{L}(B)$

(c) $\mathrm{FV}(B') \subseteq \mathrm{FV}(?)$

In order to formulate the constraint of guardedly well-definedness of recursion definitions in a behaviour expression it is necessary to introduce some concepts.

Let B and F be two behaviour expressions. Then we define *F occurs directly unguarded in B* as follows:

F occurs direcly unguarded in F.

If F occurs direcly unguarded in C then it does also in $B = C \setminus \alpha$.

If F occurs direcly unguarded in C or D then it does also in $B = C + D$, $B = C \mid D$ and $B =$ **if** f **then** C **else** D.

Proposition 17 *If F occurs direcly unguarded in in B and B is a program then F is a program also.*

We say that $\mathrm{fix}_i \tilde{X}\tilde{p}\tilde{E}$ *is directly in* $\mathrm{fix}_j \tilde{X}'\tilde{p}'\tilde{E}'$ iff for some $\tilde{e}$

either $\mathrm{fix}_i \tilde{X}\tilde{p}\tilde{E}(\tilde{e})$ does itself occur directly unguarded in E'_j

or $X_i(\tilde{e})$ does occur directly unguarded in E'_j

Now for the constraint. To shorten notation denote $\mathrm{fix}_{i_j}\tilde{X}(j)\tilde{p}(j)\tilde{E}(j)$ by $F(j)$. Let $F(1),\ldots,F(n)$ be the indexed recursion definitions in B. We then define

B to be *guardely well-defined* iff there is no infinite sequence

$$F(j_1), F(j_2), \ldots, F(j_k), \ldots$$

such that for each k, $F(j_{k+1})$ is directly in $F(j_k)$.

The constraint is then that behaviour expressions shall be guardedly well-defined. By this it should be clear that if two behaviour expressions are in an atomic action relation then it has a finite proof.

The following theorem establish an important connection between substitution, conversion and the action relations:

Theorem 18

(a) $C \xrightarrow{\Gamma} C' \Rightarrow [\delta/\gamma]C \xrightarrow{[\delta/\gamma]\Gamma} C'' \;\mathsf{cnv}\; [\delta/\gamma]C'$

(b) The following diagram commutes in the sense that if $B_0 \;\mathsf{cnv}\; B_1 \xrightarrow{\Gamma} B_1'$ *then* B_0' *exists such that* $B_0 \xrightarrow{\Gamma} B_0' \;\mathsf{cnv}\; B_1'$ *and vice versa:*

$$\begin{tikzcd} B_0 \arrow[r, phantom, "\mathsf{cnv}"] \arrow[d, "\Gamma"] & B_1 \arrow[d, "\Gamma"] \\ B_0' \arrow[r, phantom, "\mathsf{cnv}"] & B_1' \end{tikzcd}$$

(c) $[\delta/\gamma]C \xrightarrow{\Gamma} D$ *implies that there exists ?′ and* D' *such that*

$$C \xrightarrow{\Gamma'} D' \text{ and } [\delta/\gamma]D' \;\mathsf{cnv}\; D, [\delta/\gamma]?' = ?.$$

Notice that (a) and (c) in a way are each others reverses.

Proof (a) and (b) are proved by a rather complicated induction proof with hypothesis:

If $\mathcal{C}[\,]$ is a context and $B_0 \;\mathsf{cnv}\; B_1$, then

1) $\mathcal{C}[B_i] \xrightarrow{\Gamma} B_i' \rightarrow \mathcal{C}[B_{i\oplus 1}] \xrightarrow{\Gamma} B_{i\oplus 1}' \;\mathsf{cnv}\; B_i'$

2) $\mathcal{C}[B_i] \xrightarrow{\Gamma} B_i' \rightarrow [\delta/\gamma]\mathcal{C}[B_i] \xrightarrow{[\delta/\gamma]\Gamma} B_i'' \;\mathsf{cnv}\; [\delta/\gamma]B_i'$

and where the induction involves recursion depth of $\mathcal{C}[B_i]$, structure of $\mathcal{C}[\,]$, and the length of the inferences which ensures "$\xrightarrow{\Gamma}$" and "cnv". (a) and (b) then follows as a special case: $\mathcal{C}[\,] = [\,]$. (c) is proved by induction on the length of the inferences which ensures "$\xrightarrow{\Gamma}$", and uses (a) and (b) $\square$

4.1 Pushdown Store

Now where we have presented atomic action relation we will study an example of a pushdown store presented in [Mil1].

Let V be a value set, for instance integers, with $\bar{a}, \bar{b} \in V$ and variables $\bar{x}, \bar{y}$. Then let $t \in V^*$ — the set of sequences over V. $-$ is the empty sequence.As prefixing operation over V^* is used ':'. The characteristics of first,rest: $V^* \rightarrow V^*$ is $\mathrm{first}(\bar{a} : t) = \bar{a}$, $\mathrm{rest}(\bar{a} : t) = t$, $\mathrm{first}(-) = \mathrm{rest}(-) = -$.

We expect the pushdown store to be accessed through ι, o where values are pushed <u>i</u>n via ι and popped <u>o</u>ut via o. Furthermore an empty pushdown store deliver a $\$ \notin V$ when popped to indicate emptiness.

If t_x is a variable over V^* a suggestion is:

$$\begin{aligned} PD = \mathrm{fix}\, X\langle t_x\rangle\langle \iota?\bar{x}.X(\bar{x} : t_x) + \\ \textbf{if } t_x \neq - \textbf{ then } o!\mathrm{firs}t(t_x).X(\mathrm{rest}(t_x)) \textbf{ else } o!\$.X(-)\rangle \end{aligned}$$

Clearly $PD(t)$ shall satisfy:

$$PD(-) \approx \iota?\bar{y}.PD(\bar{y} : -) + o!\$.PD(-) \tag{1}$$
$$PD(\bar{a} : t) \approx \iota?\bar{y}.PD(\bar{y} : \bar{a} : t) + o!\bar{a}.PD(t) \tag{2}$$

We want a simple implementation **PD** of PD which satisfy (1) and (2).

The pushdown store will consist of some elements **E** and a bottom **B**. We extend $\bar{x}$ to range over $V \cup \{\$\}$. Then these can be specified by:

$$\begin{aligned} &\mathbf{B} = \mathrm{fix}\, X\langle(\mathbf{E}(\$, \beta_0) \mid \beta_0?.X) \setminus \beta_0\rangle, \qquad \text{where} \\ &\mathbf{E} = \mathrm{fix}\, X\langle\bar{x}, \mathsf{x}\rangle\langle\iota?\bar{y}.(X(\bar{y}, \alpha) \mid \alpha?.X(\bar{x}, \mathsf{x})) \setminus \alpha + o!\bar{x}.\mathsf{x}!.\mathrm{nil}\rangle \end{aligned}$$

(assuming ι, o, λ, α and β_0 are mutually different).

The idea is that when the top element emits it's value by a pop operation it vanish after signalling to the element just below. The element receiving the signal then knows that it is the top element. If the top element instead receives a value by a push operation it creates a new top element with the value and a link to the old top element for signalling. At first we look at an example of how 5 and 8 is pushed via ι, 8 is popped via o, and how the configuration develops by that. A pop operation on an empty pushdown store is studied too. For the $\sim$ equalities we refer to the next section.

$\mathbf{B} \overset{o!\$}{\rightarrow} (\beta_0!.\mathrm{nil} \mid \beta_0?.\mathbf{B}) \setminus \beta_0 \overset{\tau}{\rightarrow} (\mathrm{nil} \mid \mathbf{B}) \setminus \beta_0 \sim \mathbf{B}$

$\downarrow^{\iota\Gamma'5'}$

$((\mathbf{E}(5, \beta_1') \mid \beta_1'?.\mathbf{E}(\$, \beta_0)) \setminus \beta_1' \mid \beta_0?.\mathbf{B}) \setminus \beta_0$, for a $\beta_1' \neq \beta_0, \iota, o$

$\downarrow^{\iota\Gamma'8'}$

$(((\mathbf{E}(8, \beta_2') \mid \beta_2'?.\mathbf{E}(5, \beta_1')) \setminus \beta_2' \mid \beta_1'?.\mathbf{E}(\$, \beta_0)) \setminus \beta_1' \mid \beta_0?.\mathbf{B}) \setminus \beta_0$, for a $\beta_2' \neq \beta_1', \iota, o$

$\downarrow_{o!8}$
$(((\beta_2'!.\mathrm{nil} \mid \beta_2'?.\mathbf{E}(5,\beta_1')) \setminus \beta_2' \mid \beta_1'?.\mathbf{E}(\$,\beta_0)) \setminus \beta_1' \mid \beta_0?.\mathbf{B}) \setminus \beta_0$
$\downarrow_{\tau}$
$(((\mathrm{nil} \mid \mathbf{E}(5,\beta_1')) \setminus \beta_2' \mid \beta_1'?.\mathbf{E}(\$,\beta_0)) \setminus \beta_1' \mid \beta_0?.\mathbf{B}) \setminus \beta_0$
$\sim$
$((\mathbf{E}(5,\beta_1') \mid \beta_1'?.\mathbf{E}(\$,\beta_0)) \setminus \beta_1' \mid \beta_0?.\mathbf{B}) \setminus \beta_0$

Now let us define $t_n = (\bar{a}_n : \ldots : \bar{a}_1), \bar{a}_i \in V$ thereby $t_n \in V^*(t_0 is -)$ and define

$$\mathbf{PD}(t_n) = (\mathbf{E}(\bar{a}_n,\beta_n)|\beta_n?.\mathbf{E}(\bar{a}_{n-1},\beta_{n-1}) \mid \ldots \\ |\beta_2?.\mathbf{E}(\bar{a}_1,\beta_1) \mid \beta_1?.\mathbf{E}(\$,\beta_0) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0,$$

where $\beta_i \neq \beta_j$ for $i \neq j$ and $\beta_i \notin \{\iota, o, \lambda\}$.

And define:

$$\mathbf{PD}(t_0) = \mathbf{PD}(-) = \mathbf{B}$$

To write things short, denote (for $n \geq 0$):

$$\beta_{n+1}?.\mathbf{E}(\bar{a}_n,\beta_n) \mid \beta_n?.\mathbf{E}(\bar{a}_{n-1},\beta_{n-1}) \mid \ldots \mid \beta_2?.\mathbf{E}(\bar{a}_1,\beta_1) \mid \beta_1?.\mathbf{E}(\$,\beta_0),$$

where $\beta_i \neq \beta_j (i \neq j)$ and $\beta_i \notin \{\iota, o, \lambda\}$, by $\mathbf{EE}(\beta_{n+1} : t_n)$. $\mathbf{EE}(\beta_1, t_0)$ will thereby denote $\beta_1?.\mathbf{E}(\$,\beta_0)$

Then $\mathbf{PD}(\bar{a}_n, t_{n-1})$ can be written:

$$(\mathbf{E}(\bar{a}_n,\beta_n) \mid \mathbf{EE}(t_n,\beta_{n-1}) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0$$

We will prove that $\mathbf{PD}(t)$ satisfy (1) and (2) in the section about observational equivalence.

5 Strong Equivalence

The notion of strong equivalence between programs will now be presented along the lines of the definition in the original version of CCS, i.e. in terms of a decreasing sequence of equivalence relations. In later versions another definition called strong bisimulation is used instead. One of the arguments is that it admits an elegant proof technique. In spite of this we have found it difficult, to use strong bisimulation in this framework. The reason is that the proofs using strong bisimulation rely on the possibility to regard the result of a derivation as being on the same form as the premise of the derivation. By Com$\rightarrow$(3) and Res$\rightarrow$(2) this is not always the case here.

Definition 19 (Strong equivalence between programs)

$B_0 \sim B_1$ *iff* $\forall k \geq 0 : B_0 \sim_k B_1$, *where* $B_0 \sim_k B_1$ *is defined:*

$$
\begin{array}{l}
B_0 \sim_0 B_1 \textit{ is always true} \\
B_0 \sim_{k+1} B_1 \textit{ iff for all } ? \, (i = 0,1)\textit{:} \\
\textit{i) } B_i \overset{\Gamma}{\to} B'_i \Rightarrow \exists B'_{i\oplus 1} : B_{i\oplus 1} \overset{\Gamma}{\to} B'_{i\oplus 1} \textit{ and} \\
\qquad\qquad\qquad\qquad B'_i \sim_k B'_{i\oplus 1}, \qquad \textit{if } ? = \lambda!v \\
\qquad\qquad \forall v : [v/x]B'_i \sim_k [v/x]B'_{i\oplus 1}, \quad \textit{if } ? = \lambda?x
\end{array}
$$

□

Notes:

i) Each v must, in accordance with earlier notes, be comparable to x in the case $? = \lambda?x$.

ii) When B_0 and B_1 are programs it is ensured that only programs are considered in the future actions, since if B_j is a program ($\mathrm{FV}(B_j) = \emptyset$) and $? = \lambda?x$ then $\mathrm{FV}(B'_j) \subseteq \{x\}$ by proposition 16.c. By the substitution $[v/x]$ a possible occurrence of x in B'_j vanish (becomes v).

The second note gives occasion for:

Proposition 20 *In definition 19 above* $B'_j(j = 0,1)$ *is a program if* $? = \lambda!v$ *and otherwise* $[v/x]B'_j$ *is a program for all* v *comparable to* x.

If one interpret $[v/x]B'$ or B' (according to $?$) as the result of a derivation it follows from proposition 20 that the program property is invariant under derivation. The atomic action relation together with definition 19 thereby gives meaning to the programs.

Below we state a collection of simple but important properties of strong equivalence.

Theorem 21

(a) $\sim$ *is a equivalence relation*

(b) $B_0 \sim_{k+1} B_1$ *implies* $B_0 \sim_k B_1$

(c) $B_0 \sim B_1$ *implies* $B_0 \sim_k B_1$ *for all* k

(d) $B_0 \;\mathsf{cnv}\; B_1$ *implies* $B_0 \sim B_1$

Notes:

i) If we take strongly equivalence among programs as an expression for equal behaviour then (d) states that the requirement of behaviourly independence of actual names of bound elements is met.

ii) (d) has another important consequence, namely that all the results of the theory of conversion and substitution (section 3) can be lifted to strong equivalence.

Proof (a), (b) and (c) as in [Mil1]. (d) by induction on k in the definition of $\sim$ using theorem 18.b together with the convertibility properties. □

The following theorem is perhaps the hardest to accept. It says that $\sim$ is a congruence with respect to substitution of labels, i.e. two strongly equivalent programs will remain strongly equivalent if one changes the name of a free occuring label. As long as the new name is chosen different from other free occuring labels this should be clear. But it is much more difficult to convince oneself if the new name occurs in one of the programs, because it then no longer is a relabelling in the sense of the original CCS. Later versions of CCS, e.g. in [Mil3] has left out the constraint (function bijective) imposed on the first version of relabelling.

Theorem 22

$$B_0 \sim B_1 \textit{ implies } [\delta/\gamma]B_0 \sim [\delta/\gamma]B_1$$

Proof By induction on k in the definition of $\sim$ using theorem 18. This and the following proofs depends strongly on theorem 21.d and the convertibility properties. □

The next theorem states that $\sim$ is a congruence relation.

Theorem 23

$$[v/x]B_0 \sim [v/x]B_1 \textit{ for all } v \textit{ implies } \lambda?x.B_0 \sim \lambda?x.B_1$$

$$\begin{array}{lll} B_0 \sim B_1 \textit{ implies} & \lambda!v.B_0 \sim \lambda!v.B_1 & \tau.B_0 \sim \tau.B_1 \\ & B_0 + C \sim B_1 + C & C + B_0 \sim C + B_1 \\ & B_0 \mid C \sim B_1 \mid C & C \mid B_0 \sim C \mid B_1 \\ & B_0 \setminus \alpha \sim B_1 \setminus \alpha & \end{array}$$

Proof Same techniques as in [Mil1] — with some care in the order of proofs of individual properties. □

If the ECCS shall be useful, the algebraic laws of CCS must be preserved. We therefore state and prove the following theorem which collect the algebraic properties found in the original CCS. In the theorem g will stand for a guard, i.e. τ, $a!e$ or $a?x$. It will turn out that the summation operator $+$ is commutative and associative. In the light of this it makes sense to define a *sum of guards* as

$$\Sigma\{B_i; i \in I\}$$

where I is a finite index set. Each B_i is called a *summand* (s. for short).

Theorem 24 (for programs)
Sum$\sim$

(1) $B_0 + B_1 \sim B_1 + B_0$ *(2)* $B_0 + (B_1 + B_2) \sim (B_0 + B_1) + B_2$

(3) $B + \mathrm{nil} \sim \mathrm{nil}$ *(4)* $B + B \sim B$

Com$\sim$

(1) $B_0 \mid B_1 \sim B_1 \mid B_0$ *(2)* $B_0 \mid (B_1 \mid B_2) \sim (B_0 \mid B_1) \mid B_2$

(3) $B \mid \mathrm{nil} \sim B$ *(4)* *If* B_0, B_1 *are sums of guards then* $B_0 \mid B_1 \sim$

$$\begin{array}{l}\sum\{g.(B_0' \mid B_1); g.B_0' \textit{ a summand of } B_0\} + \\ \sum\{g.(B_0 \mid B_1'); g.B_1' \textit{ a summand of } B_1\} + \\ \sum\{\tau.(B_0' \mid [v/x]B_1'); \lambda!v.B_0' \textit{ a s. of } B_0, \lambda?x.B_1' \textit{ a s. of } B_1\} + \\ \sum\{\tau.([v/x]B_0' \mid B_1'); \lambda?x.B_0' \textit{ a s. of } B_0, \lambda!v.B_1' \textit{ a s. of } B_1\}\end{array}$$

Res$\sim$

(1) $(g.B) \setminus \alpha \sim \begin{cases} \mathrm{nil} & \textit{if } g = \alpha!v \\ g.B \setminus \alpha & \textit{if } \alpha \notin \mathrm{L}(g) \end{cases}$

(2) $B \setminus \alpha \sim B$, *provided* $\alpha \notin \mathrm{L}(B)$ *(3)* $B \setminus \alpha \setminus \beta \sim B \setminus \beta \setminus \alpha$

(4) $(B_0 + B_1) \setminus \alpha \sim B_0 \setminus \alpha + B_1 \setminus \alpha$ *(5)* $(B_0 \mid B_1) \setminus \alpha \sim B_0 \setminus \alpha \mid B_1 \setminus \alpha$, *provided* $\alpha \notin \mathrm{L}(B_0) \cap \mathrm{L}(B_1)$

Rec$\sim$ $\mathrm{fix}_i \tilde{X}\tilde{p}\tilde{E}(\tilde{v}) \sim [\mathrm{fix}\,\tilde{X}\tilde{p}\tilde{E}/\tilde{X}][\tilde{v}/\tilde{p}]E_i$

Con$\sim$

L: **if** true **then** B_0 **else** $B_1 \sim B_0$ *R:* **if** false **then** B_0 **else** $B_1 \sim B_1$

Remarks:

i) Remember that we are assuming the behaviour expressions to be programs when writing equations with $\sim$. As a consequence the guards in Com$\sim$(4) and Res$\sim$(1) are on the form τ, $\lambda!v$ or $\lambda?x$. Similar in Rec$\sim$ the actual parameters must be labels or values.

ii) Res$\sim$(1) gives no equivalences for the guard $g = \lambda!\alpha, \lambda \neq \alpha$.

Proof Same techniques as in [Mil1]. Order of individual proofs important. $\square$

We now want to extend the definition of $\sim$ to arbitrary comparable behaviour expressions. The reason is that we want to lift the previous results to arbitrary subexpressions of a program in such a way that we for instance can replace $B \mid \mathrm{nil}$ by B anywhere in a program.

We say that two behaviour expressions E *and* E' *are comparable* iff

> for each recursion variable X in $\mathrm{FV}(E) \cap \mathrm{FV}(E')$, X occur with comparable parameters in in E and E', i.e. if $X(\tilde{e})$ occurs in E and $X(\tilde{e}')$ occurs in E' then $\tilde{e}$ and $\tilde{e}'$ must be comparable.

If this constraint is not met for E and E' on can in advance see that it makes no sence to replace E by E' in $\mathrm{fix}\, X\tilde{p}E(\tilde{v})$.

Definition 25 *Let* $\tilde{X}$ *be the recursion varables of* $\mathrm{FV}(E_0) \cup \mathrm{FV}(E_1)$ *and* $\tilde{x}$ *the remaining variables. Then*

$$E_0 \sim E_1 \textit{ iff for all } \tilde{M}, \tilde{v} : [\tilde{M}/\tilde{X}][\tilde{v}/\tilde{x}]E_0 \sim [\tilde{M}/\tilde{X}][\tilde{v}/\tilde{x}]E_1, \qquad (3)$$

where $\mathrm{FV}(\tilde{M}) = \emptyset$ *and* $M_i = \mathrm{fix}_{j_i}\, \tilde{X}(i)\tilde{p}(i)\tilde{E}(i)$ *an indexed recursion definition such that* $\tilde{p}(i)$ *is comparable with* $\tilde{e}$ *for a occurrence* $X_{j_i}(\tilde{e})$ *in* E_0 *or* E_1. $\square$

By the way $\tilde{M}$ and $\tilde{v}$ are chosen above we always obtain programs in (3).

Up till now we have most used B for programs though it could denote a behaviour expression. The reason we have used E here is to emphasize that the definition now includes expressions as well. We state the generalization:

Theorem 26

(a) theorem 21, theorem 22 and theorem 23 holds for comparable behaviour expressions when $\lambda?x$, $\lambda!v$ *is exchanged with* $a?x$, $a!e$.

(b) Similar theorem 24 holds with the guards $\lambda?x$, $\lambda!v$ *exchanged with* $a?x$, $a!e$ *and actual parameters (in Rec$\sim$) may be* $\tilde{e}$. *Com$\sim$(4) shall be adjusted with the following condition: in the first (second) term on the right hand side of* $\sim$ *no free variable in* B_1 *(*B_0*) is bound by* g. *The other conditions of theorem 24 remains the same except* $g = \alpha!v$ *of Res$\sim$(1) which becomes* $\alpha!e$, *and* true, false *of Con$\sim$ which may be* f.

Proof Essential as in [Mil1], but with some additional difficulties. E.g. $E_0 \setminus \alpha \sim E_1 \setminus \alpha$ may look like $E_0' \setminus \beta \sim E_1' \setminus \gamma$ upon substitution of α for a free variable in E_0 and E_1. □

By Res$\sim$ it makes sense to write $B \setminus A$ if A is a finite set of labels, so we can state the expansion theorem as:

Theorem 27 (The Expansion Theorem)
Let A be a finite set of labels and B_i a sum of guards, such that if $a!\alpha.B_i'$ is a summand of B_i then $\alpha \notin A$. Then for $B = (B_1 \mid \ldots \mid B_n) \setminus A$ we have

$$\begin{aligned} B \sim & \textstyle\sum\{g.(B_1 \mid \ldots \mid B_i' \mid \ldots \mid B_n) \setminus A; g.B_i' \text{ a s. of } B_i, \mathrm{L}(g) \cap A = \emptyset\} + \\ & \textstyle\sum\{\tau.(B_1 \mid \ldots \mid [e/x]B_i' \mid \ldots \mid B_j' \mid \ldots \mid B_n) \setminus A; a?x.B_i' \text{ a s. of } B_i, \\ & \qquad\qquad\qquad\qquad\qquad\qquad a!e.B_j' \text{ a s. of } B_j, i \neq j\} \end{aligned}$$

provided in the first term on the right hand side of $\sim$ no free variable in B_k, $k \neq i$ is bound by g.

Remarks:

i) The assumption that if $a!\alpha.B_i'$ is a summand of B_i then $\alpha \notin A$ is there to avoid terms on the right hand side of the form: $\sum\{(a!\alpha.(B_1 \mid \ldots \mid B_i' \mid \ldots \mid B_n) \setminus A - \{\alpha\}) \setminus \alpha; g.B_i'$ a s. of B_i such that $a \notin A$, $\alpha \in A\}$.

ii) Notice that if a is not a label and $v \notin A$ then $\mathrm{L}(a!v) \cap A = \mathrm{L}(a?x) \cap A = \emptyset$.

Proof Essential as in [Mil1]. □

In the next section we introduce a wider equivalence relation over behaviour expressions called observational equivalence ($\approx$). We will end this section with a theorem which is as important for the connexion between $\sim$ and $\approx$ as theorem 18 of section 4 was for the connexion between `cnv` and $\sim$.

Theorem 28 *Strong equivalence "satisfies it's definition". I.e.*

$$\begin{aligned} B_0 \sim B_1 \text{ iff for all } ?\ (i = 0,1)\text{: i) } B_i \xrightarrow{\Gamma} B_i' \Rightarrow \exists B_{i\oplus 1}' : B_{i\oplus 1} \xrightarrow{\Gamma} B_{i\oplus 1}' \text{ and} \\ B_i' \sim B_{i\oplus 1}', \qquad \text{if } ? = \lambda!v \\ \forall v : [v/x]B_i' \sim [v/x]B_{i\oplus 1}', \quad \text{if } ? = \lambda?x \end{aligned}$$

Proof Essential as in [Mil1] and does also depend critically on the assumption of guardedly well-definedness, but there is a further difficulty which derives from our notion of convertibility as inherited in e.g. Com$\rightarrow$(3). The problem is overcomed by concerning equivalence classes under `cnv`. □

6 Observational Equivalence

The idea of introducing observational equivalence is to define a relation for "equal behaviours" where two programs cannot be distinguished by their ability to perform τ-actions. The τ-actions thereby gets internal and unobservable. The observable actions are then $\lambda!v$ and $\lambda?x$.
A series of actions can according to definition 1 of section 5 be written as:

$$B_0 \overset{\Gamma_1}{\to} *B_1 \overset{\Gamma_2}{\to} *B_2 \to \ldots\ldots\ldots \overset{\Gamma_n}{\to} *B_n$$

and for short

$$B_0 \xrightarrow{\Gamma_1.\Gamma_2 \ldots\ldots\ldots \Gamma_n} B_n,$$

where each $*B_i$ is B_i if $?_i = \lambda!v$ and $[v_i/x]B_i$ for some v_i if $?_i = \lambda?x$.

We abbreviate

$$B \overset{\tau^k}{\to} B' \ (k \geq 0) \text{ by } B \overset{\varepsilon}{\Rightarrow} B'$$

$$B \xrightarrow{\tau^i.\Gamma.\tau^k} B' \ (j, k \geq 0) \text{ by } B \overset{\Gamma}{\Longrightarrow} B'$$

and generally

$$B \xrightarrow{\tau^{k_0}.\Gamma_1.\tau^{k_1} \ldots\ldots\ldots \Gamma_n.\tau^{k_n}} B' \text{ by}$$

$$B \overset{\Gamma_1 \ldots\ldots\ldots \Gamma_n}{\Longrightarrow} B' \text{ or if } s = ?_1 \ldots\ldots\ldots\ldots ?_n : B \overset{s}{\Rightarrow} B'.$$

Notice that the intermediate programs are on the form $*B_i$ according to $?_i$.

Definition 29 (Observational Equivalence ($\approx$) for programs)

$B_0 \approx B_1$ *iff for all* $k \geq 0 : B_0 \approx_k B_1$, *where* $B_0 \approx_k B_1$ *is defined by*

$B_0 \approx_0 B_1$ *is always true*
$B_0 \approx_{k+1} B_1$ *iff for all s:*
i) $B_i \overset{s}{\Rightarrow} B'_i \Rightarrow B_{i\oplus 1} \overset{s}{\Rightarrow} B'_{i\oplus 1}$ *and* $B'_i \approx_k B'_{i\oplus 1}$

The v_j*'s in the intermediate* $*B_{i_j}$*'s and* $*B_{i\oplus 1_j}$*'s shall of course be the same.*
$\square$

We extend the definition of $\approx$ to include arbitrary comparable behaviour expressions as we did for $\sim$.

One of the main differences between $\sim$ and $\approx$ is captured in the following proposition:

Proposition 30

$$B \approx \tau.B$$

Proof As in [Mil1]. □

Another difference is that $\approx$ is not a congruence relation.

Proposition 31

$$\approx \text{ \emph{is a equivalence relation.}}$$

Proof Easily seen from the definition of $\approx$. □

The strong equivalence relation is contained in observational equivalence:

Theorem 32

$$B_0 \sim B_1 \Rightarrow B_0 \approx B_1.$$

Proof Similar to the one in [Mil1]. □

These results are enough to prove the desired property of the pushdown store. We have not considered the properties of observational equivalence any further.

6.1 Pushdown Store (continued)

We will now take up the pushdown store example from the section about derivation. We revive the definitions.

$$\begin{aligned} \mathbf{B} &= \text{fix}\, X\langle(\mathbf{E}(\$,\beta_0) \mid \beta_0?.X) \setminus \beta_0\rangle, \qquad \text{where} \\ \mathbf{E} &= \text{fix}\, X\langle\bar{x},\mathsf{x}\rangle\langle\iota?\bar{y}.(X(\bar{y},\alpha) \mid \alpha?.X(\bar{x},\mathsf{x})) \setminus \alpha + o!\bar{x}.\mathsf{x}!.\text{nil}\rangle \end{aligned}$$

(ι, o, λ, α and β_0 are mutually different)

We defined $t_n = (\bar{a}_n : \ldots : \bar{a}_1)$, $\bar{a}_i \in V$ whereby $t_n \in V^*$ (t_0 is $-$) and defined

$$\begin{aligned} \mathbf{PD}(t_n) = (&\mathbf{E}(\bar{a}_n,\beta_n) | \beta_n?.\mathbf{E}(\bar{a}_{n-1},\beta_{n-1}) \mid \ldots \\ &|\beta_2?.\mathbf{E}(\bar{a}_1,\beta_1) \mid \beta_1?.\mathbf{E}(\$,\beta_0) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0, \end{aligned}$$

where $\beta_i \neq \beta_j$ for $i \neq j$ and $\beta_i \notin \{\iota, o, \lambda\}$ and

$$\mathbf{PD}(t_0) = \mathbf{PD}(-) = \mathbf{B}$$

Furthermore we denoted (for $n \geq 0$):

$$\beta_{n+1}?.\mathbf{E}(\bar{a}_n,\beta_n) \mid \beta_n?.\mathbf{E}(\bar{a}_{n-1},\beta_{n-1}) \mid \ldots \mid \beta_2?.\mathbf{E}(\bar{a}_1,\beta_1) \mid \beta_1?.\mathbf{E}(\$,\beta_0),$$

where $\beta_i \neq \beta_j (i \neq j)$ and $\beta_i \notin \{\iota, o, \lambda\}$, by $\mathbf{EE}(\beta_{n+1} : t_n)$, and $\mathbf{EE}(\beta_1, t_0)$ denoted $\beta_1?.\mathbf{E}(\$, \beta_0)$, wherefore $\mathbf{PD}(\bar{a}_n : t_{n-1})$ could be written:

$$(\mathbf{E}(\bar{a}_n, \beta_n) \mid \mathbf{EE}(t_n, \beta_{n-1}) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0$$

The property we wish to prove is:

$$\begin{aligned} \mathbf{PD}(-) &\approx \iota?\bar{y}.\mathbf{PD}(\bar{y} : -) + o!\$.\mathbf{PD}(-) && (4) \\ \mathbf{PD}(\bar{a} : t) &\approx \iota?\bar{y}.\mathbf{PD}(\bar{y} : \bar{a} : t) + o!\bar{a}.\mathbf{PD}(t) && (5) \end{aligned}$$

Proof In doing this we will use lemma 33 which is stated and proved at the end of this example. We look at $\mathbf{PD}(t)$ in the case $t = t_n, n \geq 2$. The cases $n = 1$ and $n = 0$ follows similar.

$\mathbf{PD}(\bar{a}_n : t_{n-1}) = (\mathbf{E}(\bar{a}_n, \beta_n) \mid \mathbf{EE}(\beta_n, t_{n-1}) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0$

$\sim$ by Rec$\sim$ and theorem 23.

$((\iota?\bar{y}.((\mathbf{E}(\bar{y}, \beta'_{n+1}) \mid \beta'_{n+1}?.\mathbf{E}(\bar{a}_n, \beta_n)) \setminus \beta'_{n+1} + o!\bar{a}_n.\beta_n!.\text{nil}) \mid$
$\mathbf{EE}(\beta_n, t_{n-1}) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0$

$\sim$ by the Expansion theorem

$\iota?\bar{y}.((\mathbf{E}(\bar{y}, \beta'_{n+1}) | (\beta'_{n+1}?.\mathbf{E}(\bar{a}_n, \beta_n)) \setminus \beta'_{n+1} | \mathbf{EE}(\beta_n, t_{n-1}) | \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0 +$
$o!\bar{a}_n.(\beta_n!.\text{nil} \mid \mathbf{EE}(\beta_n, t_{n-1}) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0$

$\sim$ by the lemma, the Expansion theorem and theorem 23.

$\iota?\bar{y}.(\mathbf{E}(\bar{y}, \beta_{n+1}) \mid \beta_{n+1}?.\mathbf{E}(\bar{a}_n, \beta_n) \mid \mathbf{EE}(\beta_n, t_{n-1}) \mid \beta_0?.\mathbf{B}) \setminus \beta_{n+1} \ldots \setminus \beta_0 +$
$o!\bar{a}_n.\tau.(\text{nil} \mid \mathbf{E}(\bar{a}_{n-1}, \beta_{n-1}) \mid \mathbf{EE}(\beta_{n-1}, t_{n-2}) \mid \beta_0?.\mathbf{B}) \setminus \beta_n \ldots \setminus \beta_0$

$\sim$ by Res$\sim$, Com$\sim$.

$\iota?\bar{y}.(\mathbf{E}(\bar{y}, \beta_{n+1}) \mid \mathbf{EE}(\beta_{n+1}, t_n) \mid \beta_0?.\mathbf{B}) \setminus \beta_{n+1} \ldots \setminus \beta_0 +$
$o!\bar{a}_n.\tau.(\mathbf{E}(\bar{a}_{n-1}, \beta_{n-1}) \mid \mathbf{EE}(\beta_{n-1}, t_{n-2}) \mid \beta_0?.\mathbf{B}) \setminus \beta_{n-1} \ldots \setminus \beta_0$

$=$

$\iota?\bar{y}.\mathbf{PD}(\bar{y} : \bar{a}_n : t_{n-1}) + o!\bar{a}_n.\tau.\mathbf{PD}(t_{n-1})$

We have now shown that

$$\begin{aligned} \mathbf{PD}(-) &\sim \iota?\bar{y}.\tau.\mathbf{PD}(\bar{y} : -) + o!\$.\tau.\mathbf{PD}(-) \\ \mathbf{PD}(\bar{a} : t) &\sim \iota?\bar{y}.\tau.\mathbf{PD}(\bar{y} : \bar{a} : t) + o!\bar{a}.\tau.\mathbf{PD}(t) \end{aligned}$$

So if we can prove (6) and (7) below we are done, since $\sim \Rightarrow \approx$ and $\approx$ is transitive.

$$\begin{aligned} \iota?\bar{y}.\tau.\mathbf{PD}(\bar{y} : -) + o!\$.\tau.\mathbf{PD}(-) &\approx \iota?\bar{y}.\mathbf{PD}(\bar{y} : -) + o!\$.\mathbf{PD}(-) && (6) \\ \iota?\bar{y}.\tau.\mathbf{PD}(\bar{y} : \bar{a} : t) + o!\bar{a}.\tau.\mathbf{PD}(t) &\approx \iota?\bar{y}.\mathbf{PD}(\bar{y} : \bar{a} : t) + o!\bar{a}.\mathbf{PD}(t) && (7) \end{aligned}$$

We will do this by induction. So assume (6) and (7) holds for k. We shall then prove that it holds for $k + 1$.

Proof of (7): Let LHS (RHS) denote the left (right) hand side of (7).

1) $LHS \overset{s}{\Rightarrow} L$. We split out in three cases:

$s = \varepsilon$: Then L is LHS. But then also $RHS \overset{\varepsilon}{\Rightarrow} RHS \approx_k LHS$ by induction. So $LHS \approx_{k+1} RHS$.

$s = \iota?'\bar{b}'.s'$: and $LHS \overset{\iota\Gamma\bar{b}}{\rightarrow} \tau.\mathbf{PD}(\bar{b} : \bar{a} : t) \overset{s'}{\Rightarrow} L$. But $\tau.\mathbf{PD}(\bar{b} : \bar{a} : t) \overset{s'}{\Rightarrow} L$ by either $s' = \varepsilon$ and L is $\tau.\mathbf{PD}(\bar{b} : \bar{a} : t)$ or $\tau.\mathbf{PD}(\bar{b} : \bar{a} : t) \overset{\tau}{\rightarrow} \mathbf{PD}(\bar{b} : \bar{a} : t) \overset{s'}{\Rightarrow} L$. In both cases we have $RHS \overset{\iota\Gamma'\bar{b}'}{\rightarrow} \tau.\mathbf{PD}(\bar{b} : \bar{a} : t)$. In case $s' = \varepsilon$ we have $s = \iota?'\bar{b}'$ and then $RHS \overset{s}{\Rightarrow} \mathbf{PD}(\bar{b} : \bar{a} : t) \approx_k \tau.\mathbf{PD}(\bar{b} : \bar{a} : t)$ by proposition 29. In the latter case we have $RHS \overset{\iota\Gamma'\bar{b}'}{\rightarrow} \mathbf{PD}(\bar{b} : \bar{a} : t) \overset{s'}{\Rightarrow} L$ or equivalently $RHS \overset{s}{\Rightarrow} L$.

$s = o!\bar{a}.s'$: and $LHS \overset{o!\bar{a}}{\rightarrow} \tau.\mathbf{PD}(t) \overset{s'}{\Rightarrow} L$. Similar as above we find $RHS \overset{\iota\Gamma'\bar{b}'}{\rightarrow} \mathbf{PD}(t) \approx_k \tau.\mathbf{PD}(t)$ or $RHS \overset{s}{\Rightarrow} L$.

2) $RHS \overset{s}{\Rightarrow} R$. Cases:

$s = \varepsilon$: Then R is RHS. But $LHS \overset{\varepsilon}{\Rightarrow} LHS \approx_k LHS$ by induction.

$s = \iota?'\bar{b}'.s'$: $RHS \overset{\iota\Gamma'\bar{b}'}{\rightarrow} \mathbf{PD}(\bar{b} : \bar{a} : t) \overset{s'}{\Rightarrow} R$. Clearly $LHS \overset{\iota\Gamma'\bar{b}'}{\Rightarrow} \mathbf{PD}(\bar{b} : \bar{a} : t) \overset{s'}{\Rightarrow} R.I.e.LHS \overset{s'}{\Rightarrow} R$.

$s = o!\bar{a}.s'$: $RHS \overset{o!\bar{a}}{\rightarrow} \mathbf{PD}(t) \overset{s'}{\Rightarrow} R$. Then $LHS \overset{o!\bar{a}}{\Rightarrow} \mathbf{PD}(t) \overset{s'}{\Rightarrow} R$ so $LHS \overset{s'}{\Rightarrow} R$.

(6) is proved in a similar way as (7). □

Lemma 33 *Given $(B \setminus \alpha' \mid C) \setminus A$ then there exist an α such that*

$$(B \setminus \alpha' \mid C) \setminus A \sim ([\alpha/\alpha']B \mid C) \setminus A \cup \{\alpha\}$$

Proof If $\alpha' \notin \mathrm{L}(C)$ then Res$\sim$(5) and Res$\sim$(2) gives the result with $\alpha = \alpha'$ using theorem 7.a. In general take an α such that $\alpha' \neq \alpha$ and $\alpha \notin \mathrm{L}(B) \cup \mathrm{L}(C)$. Then we have by (α_3): $[\alpha/\alpha']B \setminus \alpha$ cnv $B \setminus \alpha'$ so from theorem 21.d and the congruence of $\sim$ we then conclude $(B \setminus \alpha' \mid C) \setminus A \sim ([\alpha/\alpha']B \setminus \alpha \mid C) \setminus A$. Since $\alpha \notin \mathrm{L}(C)$ the rest then follows as in the special case. □

7 Expressiveness

The aim of this section is top get an idea of the expressive power of ECCS. We have already seen how some problems have a natural solution using the capabilities special to ECCS. At first it is compared to the original CCS. Thereafter two more examples are studied: translation and Eratosthenes sieve.

When compared to CCS there is as already mentioned earlier a obvious lack of one operator: the postfixed relabelling operator [S] where S is a bijective function over labels. This operator is a part of the syntax of CCS.

Though the idea is to replace each label λ by $S(\lambda)$ ([Mil1, p.23]) no replacement is actually done. The relabelling becomes "active" through a special inference rule for the relabelling operator. With our generalized substitution prefix (which is not a part of the syntax) we are able to change labels directly. We have therefore omitted the operator and the corresponding inference rule. As an example of how the same effect can be obtained we look at the chaining combinator. In our framework one would write it as

$$([\delta/\beta]B_0 \mid [\delta/\alpha]B_1) \setminus \delta, \quad \delta \notin \mathrm{L}(B_i)$$

To emphasize that our substitution prefix is not a part of the syntax it should be mentioned that $[\delta/\beta]B_0$ denotes a new behaviour expression B_0' where β is replaced by δ in accordance to definition 3 of section 3.

7.1 Translation

In chapter 9 of [Mil1] a phrase-by-phrase translation of a language P into CCS is presented. In the following we will show how we by our extension of CCS can overcome the problem of translating a procedure such that it admits several concurrent activations. Furthermore it is shown how another parameter mechanism as call by reference can be translated. Though we assume the reader to be familiar with the notation and concepts introduced in that chapter, we revive some of the definitions relevant for our examples.

The values of variables from the programming language is kept in registers:

$$\mathrm{LOC} = \alpha?x.\,\mathrm{REG}(x), \text{ with}$$
$$\mathrm{REG}(y) = \mathrm{fix}\, X\langle y\rangle\langle \alpha?x.X(x) + \gamma!y.X(y)\rangle$$

LOC_Z is the register special devoted to Z by defining:

$$\mathrm{LOC}_Z = [\alpha_Z/\alpha][\gamma_Z/\gamma]\,\mathrm{LOC}$$

A value for Z is stored via α_Z and read via γ_Z. The scope of a variable is limited by restricting with

$$L_Z = \{\alpha_Z, \gamma_Z\}$$

Translations of an expression E ($[\![E]\!]$) are made such that they deliver the result via ϱ.

$$B_0 \text{ result } B_1 \text{ denotes } (B_0 \mid B_1) \setminus \varrho$$

A translated program $[\![C]\!]$ signals it's completion via δ whereafter it "dies". It is therefore convenient to define

$$\text{done} = \delta!.\text{nil}$$

and

$$B_0 \text{ before } B_1 = ([\beta/\delta]B_0 \mid \beta?.B_1) \setminus \beta$$

for a $\beta \notin \mathrm{L}(B_0) \cup \mathrm{L}(B_1)$.

A procedure in ECCS is then translated as

$$\begin{aligned}&[\![\,\text{PROC}\,G(\text{VALUE}\,X, \text{RESULT}\,Y) \text{ is } C_G]\!] = \\ &\qquad \text{fix}\,X_G\langle(\alpha_G!\lambda.(X_G \mid (LOC_X \mid LOC_Y \mid \lambda?x.\alpha_X!x.[\![C_G]\!]) \\ &\qquad\qquad \text{before } \gamma_Y?y.\lambda!y.\text{nil})) \setminus L_X \setminus L_Y)) \setminus \lambda\rangle\end{aligned}$$

and the call

$$[\![\,\text{CALL}\,G(E, Z)]\!] = [\![E]\!] \text{ result } (\varrho?x.\alpha_G?y.y!x.y?z.\alpha_Z!z.\text{done})$$

Notes:

i) As opposed to [Mil1] no special local version of G is needed for it to call itself recursively, because it restores itself even before it receives it's argument and since α_G from the restored G is accessible from inside $[\![C_G]\!]$.

ii) What happens in $\alpha_G!\lambda.$— is that the procedure by it's activation returns a communication link which is private to the caller and that particular activation. The privacy is ensured by Com$\rightarrow$(3) and the fact that no communication of that label is done later by neither the caller nor the corresponding activation of G.

We will now look at how the parameter mechanism call by reference can be translated. Assume the procedure declaration looks like PROC G(REF Z) is $C_G = P_G$. Then the translation could be:

$$\begin{aligned}[\![P_G]\!] = \text{fix}\,X_G\langle(\alpha_G!\lambda.(X_G \mid (\lambda?z_\alpha.\lambda?z_\gamma.[z_\alpha/\alpha_Z][z_\gamma/\gamma_Z][\![C_G]\!]) \\ \text{before}\lambda!.\text{nil})) \setminus \lambda\rangle\end{aligned}$$

and the call

$$[\![\,\text{CALL}\,G(Y)]\!] = \alpha_G?y.y!\alpha_Y.y!\gamma_Y.y?.\text{done}$$

Notes:

i) The idea is to connect the labels through which Y is accessed in the calling context to all the free occurrences of α_Z and γ_Z in C_G at "run-time". This is done by communicating α_Z and γ_Z from the calling context via the private λ to the activated procedure and substitute it for the variables z_α and z_γ which corresponds to the places where the formal parameter Z is accessed.

ii) The substitution $[z_\alpha/\alpha_Z][z_\gamma/\gamma_Z][\![C_G]\!]$ is OK even if there is a local declaration of Z in C_G. The local declaration of Z will occur on the form $(\ldots B \ldots) \setminus \alpha_Z \setminus \gamma_Z$ in C_G, so α_Z, $\gamma_Z \notin \mathrm{L}((\ldots B \ldots) \setminus \alpha_Z \setminus \gamma_Z)$ and by theorem 7.a we have $[z_\alpha/\alpha_Z][z_\gamma/\gamma_Z]((\ldots B \ldots) \setminus \alpha_Z \setminus \gamma_Z) = (\ldots B \ldots) \setminus \alpha_Z \setminus \gamma_Z$
So the local Z cannot access the outer formal parameter Z as we expect.

iii) Confusion with names in the calling context is avoided automatically through Com$\rightarrow$(3) and Res$\rightarrow$(2). If for instance α_Y "on it's way" from the calling context passes through it's own scope (the restriction of L_Y) it becomes a variable (Res$\rightarrow$(2)) and when it "arrives" at the destination a new label name is chosen (according to Com$\rightarrow$(3)) different from all other in the new extended scope.

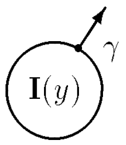
We have also made translations for call by name, but have not found it possible to do it phrase-by-phrase.

7.2 Producing Primes

We will now describe a program which emits the primes in increasing order via the label λ. We are going to use the algorithm known as Eratosthenes sieve. The program consists of two subprograms: INTEGERS and SIEVE. INTEGERS (abbreviated as **I** in the program) produce in increasing order all the integers from 2. SIEVE (**S**) sorts out the prime from the integers coming from INTEGERS. In SIEVE the subprogram FILTER (**F**) is used to filter out all multipla of a certain integer. Let p, y be variables over integers, and i, o variables over the set of labels. Then the program can be described as

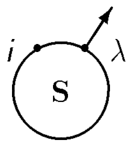
$$(\mathbf{I}(2) \mid \mathbf{S}(\gamma)) \setminus \gamma, \text{ where}$$

$$\mathbf{I} = \mathrm{fix}\, X\langle y\rangle\langle \gamma!y.X(y+1)\rangle$$



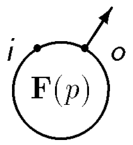
and

$$\mathbf{S} = \text{fix}\, X\langle i\rangle\langle i?p.\lambda!p.(\mathbf{F}(p,i,\alpha) \mid X(\alpha)) \setminus \alpha\rangle$$

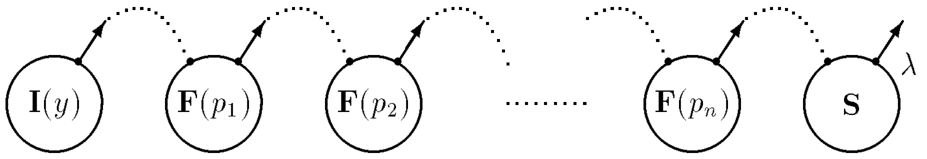


where

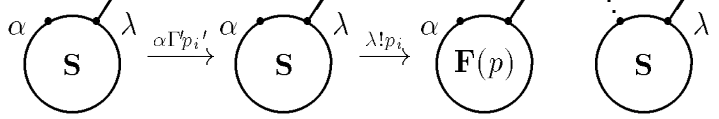
$$\mathbf{F} = \text{fix}\, X\langle p,i,o\rangle\langle i?y.\textbf{if } y \bmod p \neq 0 \textbf{ then } o!y.X(p,i,o) \\ \textbf{else } X(p,i,o)\rangle$$



A typical situation is



where P_i is the i'th prime, e.g. $p_1 = 2$, $p_3 = 5$ etc. Each $\mathbf{F}(p_i)$ gets an ascending stream of integers from it's predecessor and passes them on to it's successor, suppressing all multiples of p_i. What happens when SIEVE receives it's first integer can be demonstrated by the following derivation:



It is seen that the configuration contains as many FILTERs as primes found and each time a prime is found a new FILTER is created whereby the configuration enlarges. Notice furthermore that all FILTERs "works" concurrently in the sense mentioned in [Mil1, p.26].

8 Conclusion

ECCS as presented here is one attempt at a smooth extension of CCS satisfying the goals outlined in the introduction. In the process of defining ECCS we have considered a great number of alternatives — slowly converging to ECCS in it's present form. We feel, and we hope the reader feels the same, that ECCS is reasonably in line with the elegancy of Milners CCS — at least this has been one of our main guidelines in the process. We also fell there are some sound ideas underlying the calculus of ECCS, but we certainly do not claim that ECCS is the end product. There are still aspects with which we feel uneasy, and let us just mention a few.

One has to do with the fact that in ECCS, as presented here, only one value may be communicated at a time. As long as we are only concerned with normal data-values, it is obvious how to extend the calculus to allow tuples of values (as in CCS). But for label values the situation is not quite so obvious. At least, it requires some thought how to formulate "multi-change of scope in connection with single communication". We have chosen to present ECCS without going into these problems. Also, we have deliberately chosen not to consider the problems involved in generalizing CCS to allow passing of processes.

Another slightly unpleasing thing about ECCS is the fact that labels are somehow considered both as variables (bound by restriction) and values (to be substituted for label variables bound by input commands). Furthermore, ECCS obviously needs to be tried out on more challenging examples than the small toy problems we have considered in this paper.

Despite considerations like the above, which indicate that ECCS is maybe not yet "quite right", we are confident that ECCS represents a step on the right track in the process of solving the problems we set out to solve.

References

[AsZ] E. Astesiano, E.Zucca, "Parametric Channels via Label Expressions in CCS", Internal report, Universita di Genova, January 1984

[BAR] H. P. Barendregt, "The Lambda Calculus", Studies in logic and Foundations of Mathematics, Vol 103, North Holland,1984

[CFC] H. B. Curry, R. Feys, W. Craig, "Combinatory Logic", Vol 1, Studies in logic and Foundations of Mathematics, North Holland,1958

[Dog] T. W. Doeppner, A. Glacalone, "A Formal Description of the UNIX Operating system"

[KeS] J. R. Kennaway, M. R. Sleep,"LNET: syntax and semantics of a language for parallel processes", Internal report, University of East Anglia, Febuary 1983

[Mil1] R. Milner "A Calculus of Communicating Systems", Springer-Verlag, Lecture Notes in Computer Science 92, 1980

[Mil2] R. Milner "Calculli for Synchrony and Asynchrony", Theoretical Computer Science 25, 1983,p.267-310

[Mil3] R. Milner "Lectures on a Calculus for Communicating Systems", Springer-Verlag, Lecture Notes in Computer Science, 1984