Technical Report

UCAM-CL-TR-456
ISSN 1476-2986

Number 456

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

An evaluation based approach to process calculi

Joshua Robert Xavier Ross

January 1999

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 1998 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-456*

Abstract

Process Calculi have, starting with Milner's CCS, traditionally been expressed by specifying the operational semantics in terms of action-labelled transition relations between process expressions. Normally this has been done using transitions that are inductively defined by rules following the structure of the process expressions. This approach has been very successful but has suffered from certain problems. One of these is that the construction of weak, branching-time congruences has not been as simple as one might wish. In particular the natural weak bisimulations are not congruences, typically shown up by the introduction of summation. Secondly this method has not lent itself to the development of congruences for calculi that combine features of concurrency and higher-order functional languages. Another problem is more aesthetic. It is that in order to write these transition relations we need to use silent (τ) actions which are supposed to be unobservable. However we need to represent them explicitly and make explicit reference to them in defining the congruence relations.

In this thesis, an approach to process calculi based on evaluation to committed forms is presented. In particular two process calculi are given. The first is a first-order CCS-like calculus, NCCS. This demonstrates the possibility of giving natural weak branching-time congruences, with such features as summation, without the use of explicit silent actions. Various bisimulations are defined on NCCS, and these are related to existing equivalences for CCS. The second is a higher-order calculus, based on CML; a higher-order functional language extended with concurrent features. Again it is shown that a natural weak branching-time congruence exists. In both cases a transition relation is also given and the relationship between evaluation and transition is shown.

Contents

List of Figures

Chapter 1

Introduction

Examining the current mainstream programming languages one might be forgiven for wondering whether people can perform more than one action at once. An issue that has perplexed psychologists is whether the brain is a truly parallel "computer" or whether it merely switches between tasks so well that we cannot tell the difference. Independent of whether the brain does work in a truly parallel manner, in reality we often appear to deal with more than one situation at once. We may also work in cooperation with other people, and all of us may work at once, not requiring other people to stop while we are working. This ability to cooperate contrasts with the way current programming languages work. These languages may have been suitable while individual computers sat on a desk on their own, isolated from everything except for a printer. However as computers are now increasingly being connected together the approach needs to become one that is not so isolationist.

In his Doctoral Thesis, [Gor94], Gordon gives an implementation of a functional language with input and output. His justification for the desirability of such a language is that functional languages are easier to reason about, but input and output are needed. Input and output complicate language design, and make reasoning more difficult. His language, however, does not involve concurrency, but merely extends the functional language with primitives for input and output. To deal easily with a multiplicity of possible machines that may wish to communicate with us, we would prefer to be able to use a concurrent language. Then we could allow parts of the

program to "lie dormant" until an external source wishes to communicate with us. A part of the program could then be awakened to deal with the request, or interact with the larger program in order to deal with the request. Once such communication is possible we may also want to have the ability to split our program into completely separate segments. These parts would only be able to communicate using the same mechanisms that were used to receive the original request. Such a symmetric approach would allow easy integration of our programs into the larger environment of the network. However we may also desire the ability to communicate privately between various parts of a program, in order that external machines are not able to affect execution of the program more than we wish to allow. All these ideas suggest that we need to consider concurrent programming languages.

There is however a problem with desiring such concurrent languages. There needs to be a good theoretical basis for such languages, and while imperative languages may be hard to reason about, concurrent programs have additional problems with non-determinism. There are also problems with writing programs that are genuinely concurrent-safe. It is all too easy to write a program that appears to be safe, but is in fact fatally flawed. An example of this is the problem of a company with several telephone lines. Let us say they have six lines. Company rules state that only five lines may be used at once, so that there is always a line free in the case of an emergency. The following solution is proposed. A central counter is kept of how many lines are currently in use. If less than five are being used when a call is requested, one is added to the counter and the telephone call is started. Once the call is finished, the counter is reduced by one. There are two problems with this. The first is that two people might start telephoning at the same time. Both check that there are less than five people using a line, and both discover that there are exactly four people using the lines. They both add one to the counter and make a telephone call. The company is left without an emergency line. However as soon as the first person finishes his call the situation becomes safe again. The second problem is much worse. Again, two people try starting a call at the same time. Both read the value of the counter, and both add one to the value they "know" the counter has. Both

then update the counter to their "correct" value. The situation is now that all six lines are in use, but the counter only reads five. This situation will not remedy itself when one of the lines becomes free, and even worse it will now allow any person to accidentally block the emergency line by taking what they consider to be the fifth line. The same problem occurs with the reduction of the counter phase if two calls end simultaneously. The system could end up in a state whereby only one person could use the telephone at once!

Thus we come to the conclusion that we need some tools to help us to reason about programs, since there are so many pitfalls.

1.1 Background

Much work has been done to analyse programming languages, particularly functional ones. The λ-calculus has been studied widely as the basis of functional languages. Possibly the best known example of a large scale operational semantics is the definition of Standard ML [MTH90]. It has seemed natural to use a "big step" reduction relation for the operational semantics of functional languages. This is because purely functional languages are deterministic, because of the idea that functions take a value and return a value, and because the intermediate steps are not of interest.

In contrast to functional languages, the operational semantics of languages for concurrent communicating processes have traditionally been specified by action-labelled transition relations between process expressions. These transition relations usually include a "silent action", often denoted τ. Ideally, these transition relations have been inductively defined by rules following the structure of expressions. There are various reasons for using a transition relation. One of these is that concurrent languages do not operate on the model of taking a value and returning a value. In general their programs may never terminate, but instead continue to interact indefinitely with the environment.

1.1.1 Evaluation vs Transition

We have two different approaches to designing reduction relations. The first, from concurrent communication processes, is that of using a "small step" reduction relation or *transition* relation. The second, from functional languages, is that of using a "big step" reduction relation or *evaluation* relation. Since there are differences between evaluation and transition relations we need to examine the advantages and disadvantages of the two approaches.

The most obvious difference is that a transition relation is more fine grained than an evaluation relation. This may be viewed as either an advantage or a disadvantage depending on your point of view. If we want to consider the overall position of a program then we want to deal with as little information as we can, whilst still being able to get the information about the program that we need. Typically, the more information we are presented with, the harder we find it to determine the facts that we need. We tend to "lose the forest for the trees". On the other hand, we do not want to reduce the amount of information that we have to deal with to the extent that we do not have the information that we do need.

Another difference between evaluation and transition is the use of silent actions. An important question is whether silent actions are truly silent or not. This could be considered to be a question of whether we can deduce that one or more silent actions are being performed by the time that a program takes to execute. However this suffers from the problem that it is unusual for two different "silent actions" to take the same length of time to execute. Also one might argue that one of the uses of equivalences is in optimising programs. This may be done by hand, or by a compiler that has been taught various equivalences. Using such a technique one might write a program that is simple to reason about, but use a faster optimised version to execute. An example of this is that various C compilers will transform tail, and some other forms of, recursion into a loop [SS76]. Tail recursion is sometimes an easier way to write a program but is slower to run, due to the overheads of the function call.

There is also an ideological reason to suggest that there should not be

silent actions in the language. If they are truly silent then they should not be observable. This may also be reasoned from the idea, suggested above, that actions do not all take the same time to execute. Another suggestion is that there may be other programs running on the same machine so it is not known whether the slowness is caused by the program, or if the problem results from too many programs running at once. Whatever the reason, or mixture of reasons, silent actions may not be observed. If we cannot, or refuse to, observe silent actions, then why should they be included in the reduction relation?

1.1.2 First-order calculi

The λ-calculus has formed the basis for analysing functional programming languages. In a similar way CCS, first in [Mil80] and then updated in [Mil89], has formed the basis for reasoning about concurrency. Although the communication in CCS is limited to synchronisation, it still has sufficient power for some applications. A simple example is to define the following agent, in the notation of CCS (recalling that $r.P$ means "input on channel r and then become process P"):

$$A = r.e.A$$

Returning to the problem of the company having six telephone lines, we then let the rules on using the telephone be as follows.

- Attempt to transmit on r.
- Wait for the transmit on r to be matched by an input on r, or give up and stop trying to use the telephone for now.
- Make the telephone call.
- Attempt to transmit on e, possibly spawning off another process to do this.

These are combined with an additional rule that no other reference may be made to the channels e and r.

This could be implemented as the process (remembering that $\overline{r}$ means output on channel r)

$$\overline{r}. < \text{Telephone Call} > .(\overline{e}.0 | < \text{Rest of the program} >)$$

The possibility of giving up can be introduced using the choice, or summation, operator. The number of lines available for non-emergency use is given by the number of repetitions of A in $A|A|\ldots|A$, the central (or possibly distributed) set of line controllers. This implementation may be reasoned about relatively easily, if it is assumed that everyone using the telephone system obeys the rules described above. Of course there will be problems if someone decides to abuse the system, for example by not releasing the line after use.

There have been other similar calculi. For example CSP [Hoa85], which is again based around the concept of indivisible interaction. There has also been a process algebra approach, for example ACP and ASP used in [Wei89].

Milner shows in [Mil89] that the passing of a fixed set of values can be encoded within CCS, and so does not increase the expressive power of CCS.

1.1.3 Higher-order Calculi

The question of what happens to the power of the calculus when channel names may be passed over channels lead to the creation of the π-calculus [MPW89, Mil91]. Although at first sight this does not give all that much extra power, it does in fact add as much power as we may want. For example Milner has shown in [MPW89] that numbers can be encoded in the π-calculus in the style of Church numerals. In [San92, San93a] Sangiorgi defines HOπ that allows the passing of processes, or agents. He then goes on to show that the π-calculus can be encoded in HOπ. What is slightly more surprising is that he shows that HOπ can be encoded in the π-calculus. An intuition for this may be seen as follows: A function may be implemented as a replicated process, which reads in a value, and sends out a value at the end. This is slightly complicated by the need to ensure that the reply, and other communications, go to the correct process. This may be ensured, using the mobile nature of the π-calculus, by creating a new private channel name and passing that as the

first value given to the function. All further communication may then use this private channel. For a further examination of this issue see [San93b].

All this would suggest that the π-calculus should be taken as basic and the basic building block for concurrent programming languages, in the same way that the λ-calculus has been for functional languages. However there is still the question of exactly which π-calculus should be counted as basic. There is the issue of whether it should be polyadic or monadic. It is easily shown that the polyadic π-calculus can be encoded in the monadic π-calculus [Mil91]. In [Bou92] Boudol introduces the asynchronous π-calculus and shows that is is equivalent to the synchronous π-calculus. Honda and Tokoro present a very similar asynchronous π-calculus, although with with a more complicated encoding of the synchronous π-calculus, in [HT92].

In [Wal92] Walker gives another example of the power of the π-calculus. He shows that it can encode POOL [AdBKR92, Rut92], a parallel object oriented language.

This does not mean, however, that the π-calculus is the correct language to use in all circumstances. Indeed if one was to suggest to a programmer that he should program in hexadecimal, since "it can encode the language you are currently using", he would reply that hexadecimal is not an appropriate language to work in. Similarly, using a higher-order language will allow increased clarity, and ease of reasoning than programming by encoding into a first-order language. It does however have the advantage of a large body of theoretical knowledge and a number of different approaches to equivalences.

One example of a programming language built upon the π-calculus is PICT [PT97]. Pierce and Turner describe how a programming language can be created by using a powerful type system, based on [PT94], and the asynchronous π-calculus. The asynchronous π-calculus is used to aid efficiency and the type system is used to eliminate run time errors. This language runs efficiently but is like programming in "concurrent machine code". This is one example where a minimalist approach is not necessarily the best one.

This leads us to consider explicitly higher-order calculi. Various different higher-order process calculi have been studied, including Concurrent Clean

[AP93, AP95b, AP94, AP95a], a concurrent extension of Clean which is a lazy functional language which includes I/O. It was designed to implement and interface with a Graphical User system. This shows that a concurrent extension to a functional language can create a useful language. However no theoretical foundation is given for the language Concurrent Clean.

Concurrent Haskell [JGF96] is another example of a lazy functional language which has had concurrent primitives added to provide better support for sophisticated I/O performing programs. The language is stratified into a deterministic layer and a concurrency layer and is specifically a functional language with added concurrency.

CHOCS [Tho93], in which communication always involves the sending of a process is a higher-order language which does have a theoretical foundation. It takes the different approach of extending a concurrent language with higher order primitives. HOπ [San93a], in which both processes and names may be sent also has a good theoretical foundation. Sangiorgi also gives various equivalences for HOπ. Again it is created by extending a concurrent language to include higher-order primitives. As a result of being extensions of concurrent languages neither CHOCS nor HOπ have functions explicitly. This is not a problem for the π-calculus and its derivatives as discussed above.

CML [Rep92, FHJ95, BMT92] is a calculus obtained by extending ML with concurrency and communication primitives. It has functions, as well as names and delayed expressions, as first class values. Any value may be transmitted over channels. There has been some work done on showing that CML is a useful language to work with, in particular by Reppy [Rep92]. Similarly various different aspects of the theoretical foundation for CML have also been studied. Just as Reppy uses CML for practical utility, Berry, Milner and Turner use it in [BMT92] because of its simple semantics. This suggests that this set of primitives is natural. They go on to show that the semantics preserves the behaviour of sequential expressions.

In [FHJ95] Ferreira, Hennessy and Jeffrey give versions of core CML called μCML and μCML$^+$. These however do not include dynamic channel name creation (or private or restricted channel names). This is a serious limitation

since many of the encodings of one version of a language into another require dynamic name creation. It may be argued that since CML already has higher-order functions and may transmit functions, names and delayed expressions that this does not matter. However the encoding of, for example, an object oriented approach to programming requires dynamic names to be natural. The addition of dynamic name creation is, unfortunately, not as easy as it might appear at first sight. Similarly in [Jef98] Jeffrey defines μCMML (Concurrent monadic ML) which is a version of μCML with a much more explicit order of evaluation. The argument used is that this simplifies the semantics and gives better equational properties. Any μCML expression may be translated into μCMML. Again there is no dynamic channel name creation.

In [NN95] Neilson and Neilson discuss strategies for allocating processes to processes in a concurrent higher-order programming language such as CML or FACILE, with particular focus in this case on CML. They consider how best to match the network configuration to the communication topology of the program. The two methods are static allocation, or allocation at compile time, and dynamic allocation, or allocation at run time. The former method is used in the current implementations of FACILE, although the second, which is developed in the paper, gives a finer grained control.

Another higher-order concurrent functional language is FACILE [GMP89, TLP+93]. This is a language which has been developed by working both from the theoretical side and from the practical programming side. The theoretical side keeps the language to one which can be given a good theoretical basis and the programming side keeps it useful. It is built upon ML and aims to maintain a symmetry between functions and processes. Each part refers to the other but doesn't subsume it. This is in contrast to our method of integrating processes and functions into one whole calculus. In [Ama95] Amadio considers the core of FACILE in a parallel way to [FHJ95] for CML. He gives the barbed bisimulation and shows, by proving the adequacy of a natural translation of FACILE into the π-calculus, that this equivalence deals with restriction in a more satisfying way. He also introduces an asychronous version of FACILE.

One significant difference between our approach to CML and the approach to the various calculi described above is that of the reduction relation. In each case the reduction relation is defined in terms of a transition, or small-step, relation for the concurrent part of the language. In our approach we show that we may use an evaluation, or large-step, relation for both the functional and concurrent aspects of the language.

1.1.4 Reduction based Equivalences

Various reduction based equivalences have been defined for CCS and CCS-like calculi. The most popular method for establishing such reduction based equivalences tends to be the use of a bisimulation relation. A bisimulation relation, $\mathcal{R}$, progresses to itself under the reduction relation. In fact such an approach is not restricted to concurrent communicating calculi. In [Gor95] Gordon describes how the bisimulation approach may be used instead of the more usual contextual equivalence for functional languages. He goes on to show the equivalence of the bisimulation and the contextual equivalence for FPC and hence that co-inductive proofs may be used to prove desired properties.

There are many different equivalences that have been suggested for concurrent communicating calculi. They may be broadly split into two groups. The first, strong equivalences, depend on being able to count the number of silent actions that a process performs in any given reduction. The second, weak equivalences, do not generally allow silent actions to be counted. They may either allow silent actions before and after visible actions, as in the case of CCS's observation equivalence, found in [Mil89], or only before the visible action, as in the case of delay bisimulation in [Wei89], but ascribed there to [Mil85]. This could be viewed as a process evaluating until it either requires some form of input or wishes to output. Such a communication could be an interaction with a person or with another process. In either case it does not continue evaluation until the desired communication has taken place.

The strong equivalences are smaller, as the name suggests, but generally have better properties of substitutivity. Weak equivalences, however, are

larger and include the idea that silent actions are invisible.

Once the communications are not restricted to synchronisations, as is the case in CCS, CSP or ACP, then the possibilities for defining different equivalences become much greater. Far more different decisions may be taken about exactly what is required for two expressions to match each other's actions.

In [MS92] Milner and Sangiorgi suggest barbed bisimulation as a possible general bisimulation. It does not depend on the particular calculus, or even on whether the calculus is first or higher-order. In the strong case, two processes that are equivalent must be able to match each other in performing a silent action, and remain equivalent. Furthermore if either can perform a visible action then the other must also be able to perform a visible action (although not necessarily the same action). It is worth noting that the visible actions are not compared, and the processes are not required to remain equivalent after the visible action. In the weak case they need only match each other in the number of visible actions they may each perform. This is too weak to be of general use and so one of two methods is used to strengthen the equivalence. The first is to quantify over all contexts, or sometimes over a restricted set of contexts. The second is to partition the set of visible actions and allow the observer to know which partition the visible action is in. Sometimes a combination of these two approaches is used[1]. However, whichever approach is used, weak barbed equivalences effectively always allow silent actions after a visible action.

Various different approaches have been suggested for the different versions of the π-calculus. One method, as in [Mil91], is to define a reduction equivalence[2] and then to define the related reduction congruence by stating that two processes are reduction congruent if they are reduction equivalent for all contexts with a single hole. This has the disadvantage of having a quantification over essentially all processes, making proofs difficult.

In [ACS96a, ACS96b] Amadio, Castellani and Sangiorgi describe various bisimulations for the asynchronous π-calculus. The first they give is a version

[1] At times these approaches are also used with a strong barbed equivalence. For example to recover strong equivalence. See [MS92].

[2] In this case using a barbed bisimulation like equivalence.

of the barbed bisimulation, in which the observable actions are those which output and only the channel name is observable. A more refined version, called $o\tau$-bisimulation, is then proposed in which output transitions, and not just the channel name of output, are observable. However both the barbed bisimulation and the $o\tau$-bisimulation are too weak. The barbed bisimulation is refined again, to the barbed equivalence, this time by quantifying over parallel contexts. A refined version of the $o\tau$-bisimulation is then presented and shown to be equivalent to barbed equivalence. It is also shown that the barbed equivalence is a congruence relation. This case has the advantage that the quantification is only over parallel contexts and not all contexts, which simplifies the proof of equivalence of pairs of processes.

In [Jef98, FHJ95] various bisimulations are given for μCML^+. These vary from requiring actions to be identical to be considered as matching to requiring only that the visible, or not silent, parts of actions be related. In each case, however, the equivalences depend on comparing actions on their own. That is the actions must be related in some way that requires comparison of channel names. This will cause problems once private channel names are used, particularly if they are dynamically created.

One possible solution to this is to use a version of the barbed bisimulation, since we may either choose to equate all actions, or to differentiate between actions on the basis of the channel on which the value was received or transmitted. Another approach, that of context bisimulation, is suggested by Sangiorgi in [San94] and revised in [San95]. This approach leads to a bisimulation that is a congruence and only discriminates between expressions that we would like to be different.

1.1.5 Other Equivalences

However, reduction based equivalences are not the only type of equivalence that are used for concurrent communicating calculi. One alternative approach is that of denotational semantics. This is used by Hennessy in [Hen96] for the π-calculus and in [Hen94] for higher-order calculi, in this case an extended version of CHOCS. In each case he shows that two processes being denotationally equal implies (and is implied by) them being behaviourally

equivalent for some behavioural preorder. The difficulty with this method is that it is not necessarily any easier to see whether two processes have the same denotation than to show that they are equivalent in the reduction based equivalence.

A different approach is given in [HL92] where Hennessy and Lin extend the idea of a transition graph to that of a symbolic transition graph. This allows the definition of symbolic bisimulations and the equivalence with a reduction based bisimulation.

1.2 Contribution of the Dissertation

The main aim of this dissertation is to examine the use of an evaluation, or big-step, approach to process calculi rather than the small-step approach that is often used with more usual presentations of process calculi. One advantage of having an evaluation based approach would be that the integration of functional and concurrent aspects of programming languages could be achieved without separating the two aspects. To this aim we present two calculi. The first, a lower-order calculus, is a version of CCS with summation restricted to guarded summation, which we call NCCS. The second is a higher-order calculus based on CML, which we call CML_ν. We consider the core of the language, but still include dynamic name creation and recursive higher-order functions. In each case both a transition and an evaluation relation are defined and the relationship between the two explored. The evaluation relation for NCCS and both reduction relations for CML_ν use a structural congruence relation in the style of CHAM [BB92] to reduce the number of rules required for the reduction relation. We show that an evaluation based approach is indeed possible.

We also define various equivalences for both NCCS and CML_ν. In each case an approach involving quantifying over various parallel contexts is used to define the bisimulations. For NCCS we show that we may give bisimulations that are equivalent to many of the standard CCS equivalences. The exception is strong equivalence which we should not expect to be able to define since the calculus implicitly ignores silent actions. We then show that

in each case the parallel contexts may be restricted further and that only very simple parallel contexts are required. For CML_ν we use an approach similar to the context bisimulation of [San95]. The differences arise in the definitions because in CML_ν values may be returned as well as being transmitted over channels. In each case we again show that we may restrict the parallel contexts required. We also show that all the bisimulations, both for NCCS and CML_ν, are congruence relations.

We have thus defined two process calculi that use an evaluation relation and have shown that bisimulation relations that are also congruence relations may be defined. We have related them to similar calculi using a transition based approach. We have also defined a symmetric evaluation based calculus for a CML like language with dynamic name creation.

For the finite fragment of NCCS we give a complete axiomatisation and use a slightly different method for proving the correctness of the axiomatisation. This method may be of use in cases where the standard method of saturation[Mil89] will not work, however the author does not know of any examples where it has this advantage.

1.3 Outline of Dissertation

The main theme of this dissertation is that of using a reduction relation that is "big step" in nature, i.e. an evaluation relation. This allows languages to be developed without the recourse to using explicit silent actions. Thus it is possible to model the concept of a process evaluating until it desires to communicate with the environment, whether the communication be with the user or another process. This usage of evaluation is linked to the idea of using a delay-like equivalence.

The evaluation based approach is examined for two very different calculi, and hence the dissertation is split into two main parts. The first deals with the lower-order calculus NCCS, which is a CCS like calculus, and the second with CML_ν, a version of CML. In each case an evaluation relation is given for the calculus. Each part is described below, and the dissertation ends with some conclusions that may be drawn from the work.

1.3.1 Part I — NCCS

In chapter 2 the calculus *Normal CCS*, or NCCS[3], is presented, together with the extended calculus of τNCCS. The latter is basically the same as CCS with summation restricted to guarded summation. The former may be viewed as the τ free fragment of CCS with summation restricted to guarded summation. An evaluation relation, based on the idea of committed forms, as suggested by Pitts, is defined for NCCS, and a transition relation, equivalent to the transition relation of CCS, is given for τNCCS. The chapter concludes with translations between the two calculi.

Chapter 3 gives various bisimulations defined for both NCCS and τNCCS. The bisimulations include both the strong and weak equivalences of CCS, and also a version of weak barbed bisimulation. In addition to these, a weak delay-like bisimulation called Evaluation Bisimulation, again suggested by Pitts, is given. The basic properties of the bisimulations are given, together with the relationship between them. This chapter may be viewed as a summary chapter since the proofs of the properties and relationships are given in chapters 4 and 6.

The proof of congruence for Evaluation Bisimulation takes the whole of chapter 4 and uses an adapted version of Howe's method [How89]. This is an extended example of using Howe's method in the concurrency context, and includes the modifications required to deal with non-determinism, the use of a bisimulation rather than a simulation, and the presence of a structural congruence relation.

The relationship between the evaluation relation for NCCS and the transition relation for τNCCS is examined in chapter 5. It is proved that evaluation very nearly corresponds to a sequence of silent actions followed by a visible action in the transition relation. The reason for the imperfect match is two-fold. The first is that a structural congruence relation is used for the evaluation relation but not for the transition relation. Therefore the bracketing of processes and the scope of the restriction operator may vary. The other reason is that a translation has to be used and, as shown in chapter 2, going from τNCCS to NCCS and back again does not necessarily

[3]The design of NCCS was joint work with my supervisor Andrew Pitts.

result in an identical process, but only in a process that is equivalent, using any equivalence given in chapter 3. The relationship between evaluation and transition is then used in chapter 6, in which the various relationships between the bisimulations are proved.

This part of the dissertation concludes with an equational axiomatisation for Evaluation Bisimulation on the finite fragment of NCCS. This axiomatisation is shown to be both sound and complete. The axiomatisations for both the strong and weak equivalences of CCS are given in [Mil89] and may be easily translated into the syntax of τNCCS. Thus we have a complete set of axiomatisations, since it has already been proved in chapter 6 that all the bisimulations defined in chapter 3 coincide with one of Evaluation Bisimulation, strong equivalence or weak equivalence.

1.3.2 Part II — CML$_\nu$

The second part, dealing with a calculus based on CML, proceeds in a similar manner to the first part. It starts in chapter 8 with the syntax of CML$_\nu$. This forms a core of CML and includes higher-order functions and concurrency constructs. The evaluation relation is then given and an example of a small CML$_\nu$ program is worked through to help understand the calculus together with the evaluation relation. To help with an understanding of CML$_\nu$ and its evaluation relation, the corresponding transition relation is given in chapter 9. This transition relation is in fact the same as the one given in [FH95] with the addition of rules for restriction and without the A (always) operator. Various results are derived, including the correspondence of evaluation with a sequence of silent actions followed by a visible action in the transition relation. In contrast to NCCS there is no mismatch between the evaluation relation and the transition relation, due to both the inclusion of the structural congruence in the transition relation, and to not having to translate between calculi.

A set of requirements for bisimulations for CML$_\nu$ are given at the beginning of chapter 10 and then a few examples of equivalent and inequivalent programs are given. An extended example of the possible complications with determining equivalences is then given, which also acts

as a further guide to writing program fragments in CML_ν. The Evaluation Bisimulation is then stated for CML_ν. The Strong Bisimulation for CML_ν is also introduced, together with its relationship to Evaluation Bisimulation. Various results required for the proof of congruence are then given. The chapter concludes with two sections. The first of which gives various example equivalences and the second considers the relationship with the weak higher-order bisimulation of [FH95]. The congruence proof for Evaluation Bisimulation takes the whole of chapters 11 and 12. This again follows Howe's method but is even more complicated than the previous adaptation.

Part I

NCCS

Chapter 2

Process syntax

In this chapter we present a subset of CCS [Mil89] which we call *normal* CCS (NCCS), together with the operational semantics. We also give an extension (τNCCS) of this calculus, which includes silent actions, together with its operational semantics, and then proceed to give translations between these two calculi. The operational semantics are compared and the relationship between them is then given in chapter 5. In chapter 3 we give various notions of equivalence, including some corresponding to known equivalences for CCS, showing that each one is a congruence.

NCCS is based on the concept of *committed forms* to which processes may reduce. It is thus natural to consider a "big step" evaluation style with processes reducing to *committed forms.* It also only allows *guarded* summation and this carries over into τNCCS, in which we allow both labels and silent actions to act as guards. At the end of section 2.3 we consider how we could extend the syntax to allow unguarded summation and comment on why we don't.

An ideal for equivalence relations for processes in NCCS would be that two equivalent processes may be swapped in programs wherever they occurred. That is, we desire that an equivalence relation should be a congruence relation. The equivalence relations we give are co-inductively defined bisimulations and are based on the idea that the primary construct in concurrent communicating processes is that of (parallel) composition. We therefore use the notion of *contextual equivalence* as the basis of the bisimulations, restricting the contexts to parallel compositions of varying

degrees of freedom. We go on to prove that these seemingly restricted conditions do indeed allow two equivalent processes to be exchanged in a program.

τNCCS, defined in section 2.4 on page 31, is essentially just CCS with summation restricted to action-prefixed summation. In section 2.5 on page 33 we give a "small step" labelled transition relation which corresponds to the labelled transition relation of CCS.

2.1 Syntax of NCCS

The syntax of NCCS is based on CCS. The differences are that there are not explicit silent actions and summation is restricted to guarded summation. The syntax for processes in NCCS is given in figure 2.1.

P	::=	X	variable
	|	PP	binary composition
	|	S	summation
	|	$\nu n.P$	restriction
	|	rec $X.P$	recursion
S	::=	$(S+S)$	binary summation
	|	G	guarded process
	|	0	null process
G	::=	$n.P$	input guarded process
	|	$\overline{n}.P$	output guarded process

where
$X \in$ VAR, countably infinite set of *variables*
$n \in$ NAME, countably infinite set of (channel) *names* (disjoint from Var).

Figure 2.1: Syntax of NCCS

Note 2.1

We may interchangeably write PQ and $P|Q$ to mean the same process. We will normally write PQ, using bracketing where necessary.

Note 2.2

For any name n then $\overline{\overline{n}} = n$ and so for any co-name $\overline{n}$ then $\overline{\overline{\overline{n}}} = \overline{n}$.

Definition 2.3

The free names of a process P, $\mathrm{fn}(P)$, are defined as follows:

$$\begin{aligned}
\mathrm{fn}(0) &= \emptyset \\
\mathrm{fn}(X) &= \emptyset \\
\mathrm{fn}(n.P) &= \{n, \overline{n}\} \cup \mathrm{fn}(P) \\
\mathrm{fn}(\overline{n}.P) &= \{n, \overline{n}\} \cup \mathrm{fn}(P) \\
\mathrm{fn}(P_1 P_2) &= \mathrm{fn}(P_1) \cup \mathrm{fn}(P_2) \\
\mathrm{fn}(\textstyle\sum P_i) &= \cup_i \mathrm{fn}(P_i) \\
\mathrm{fn}(\mathbf{rec}\ X.P) &= \mathrm{fn}(P) \\
\mathrm{fn}(\nu n.P) &= \mathrm{fn}(P) \setminus \{n, \overline{n}\}
\end{aligned}$$

The bound names, $\mathrm{bn}(P)$, are defined in a corresponding manner to be those names which are bound by the restriction operator. We have corresponding definitions for free and bound variables, $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$, with **rec** binding variables instead of ν.

Note 2.4

A name (variable) may be both free and bound in a given process. The two names are distinct since any bound name (variable) may be renamed using α-conversion.

Definition 2.5

We say that a name n is *fresh* for P if it does not appear in P, either as a free name or a bound name. We typically don't specify P when it is obvious from the context and take the most general P possible.

Note 2.6

Name restriction is written $\nu n.P$, rather than $P\backslash n$ as used in CCS. This is because we prefer to view it as a binding action: free occurrences of n in P become bound in $\nu n.P$. Also we write the CCS expression $\mathbf{fix}(X = P)$ as $\mathbf{rec}\ X.P$.

We observe that all processes in NCCS are τ-free; that is, they have no explicit silent actions.

2.2 Structural equivalence on NCCS

Starting with the "chemical abstract machine" of Berry and Boudol [BB92], structural congruences have been used to simplify the specification of operational semantics. We use one here to help reduce the number of rules required to define the operational semantics for NCCS. Also, the structural congruence identifies pairs of processes that we would expect any reasonable notion of process equivalence to identify. The associativity and commutativity of composition are used because we regard all unguarded processes as being able to take part in reductions. Similarly we regard any part of a (possibly deeply nested) summation as being available. The rules affecting restriction are present because we regard name restriction as being a static construct, not a dynamic one.

Definition 2.7

We let $\equiv_\alpha$ be the smallest congruence for NCCS containing:

$$\begin{array}{rcll} \mathbf{rec}\, X.P & \equiv_\alpha & \mathbf{rec}\, Y.P[Y/X] & Y \notin \mathrm{v}(P) \\ \nu n.P & \equiv_\alpha & \nu m.P[m/n] & m \notin \mathrm{n}(P) \end{array}$$

Definition 2.8

We let $\equiv$ be the smallest congruence for NCCS containing:

$$P \equiv Q \qquad P \equiv_\alpha Q$$

$$\begin{array}{rcl} (PQ)R & \equiv & P(QR) \\ PQ & \equiv & QP \\ P0 & \equiv & P \end{array}$$

$$\begin{array}{rcl} P+Q & \equiv & Q+P \\ P+(Q+R) & \equiv & (P+Q)+R \end{array}$$

$$\begin{array}{rcll} \nu n.(PQ) & \equiv & (\nu n.P)Q & n \notin \mathrm{fn}(Q) \\ \nu n.\nu m.P & \equiv & \nu m.\nu n.P & \end{array}$$

$$\nu n.0 \equiv 0$$

Note 2.9

We observe that the structural congruence is defined for process expressions and the contexts are processes with process holes. The structural congruence for summation is not included since it is implicit within our definition of summation.

Lemma 2.10

For any NCCS processes P and Q then $P \equiv Q$ implies that $\mathrm{fn}(P) = \mathrm{fn}(Q)$.

Proof

No rule used in the definition of structural congruence changes any free name. □

Lemma 2.11 *Substitutivity for* $\equiv$

For two (possibly open) NCCS processes P_1, P_2 with $P_1 \equiv P_2$ and a NCCS process Q then $P_1[Q/X] \equiv P_2[Q/X]$.

Proof

The result follows by induction on the derivation of $P_1 \equiv P_2$. □

2.3 Evaluation relation for NCCS

Note 2.12

If $\vec{n} = \{n_1, n_2, \ldots n_m\}$ then we write $\nu\vec{n}$ as shorthand for $\nu n_1.\nu n_2.\ldots.\nu n_m$. In particular m may be 0 in which case $\vec{n} = \emptyset$ and there are no names being restricted.

The operational semantics of NCCS are specified by the inductively defined "big step" *evaluation relation*, given in figure 2.2 overleaf, where l ranges over names, n, and co-names, $\overline{n}$.

The elements of the relation are of the form

$$P \Downarrow C$$

where P is a process and C is a committed form. The committed forms, which are given in figure 2.3 on the next page, are those processes which are committed to performing a particular input or output.

$$(\Downarrow \text{PRE}) \qquad \frac{}{\nu\vec{n}.((l.P + S)Q) \Downarrow l.\nu\vec{n}.(PQ)} \qquad (l, \bar{l} \notin \{\vec{n}\})$$

$$(\Downarrow \text{COMP}) \qquad \frac{P \Downarrow l.P' \quad Q \Downarrow \bar{l}.Q' \quad \nu\vec{n}.(P'Q') \Downarrow C}{\nu\vec{n}.(PQ) \Downarrow C}$$

$$(\Downarrow \text{REC}) \qquad \frac{\nu\vec{n}.(P[\mathbf{rec}\ X.P/X]Q) \Downarrow C}{\nu\vec{n}.((\mathbf{rec}\ X.P)Q) \Downarrow C}$$

$$(\Downarrow \text{STRUC}) \qquad \frac{P \equiv Q \quad Q \Downarrow l.Q' \quad Q' \equiv P'}{P \Downarrow l.P'}$$

Figure 2.2: Evaluation relation for NCCS

$$\begin{aligned} C &= n.P \\ &\mid \overline{n}.P \end{aligned}$$

Figure 2.3: Committed forms for NCCS

Note 2.13

One effect of using this "big-step" evaluation relation is that we must have an input or an output to create a committed form. This means that $\nu a.(\overline{a}.0|a.0)$ cannot commit to anything since it cannot input or output on channel a, and 0 is not a committed form.

The evaluation rules need some explaining since there are other, possibly more obvious, rules that we might want to use. We use the ($\Downarrow$STRUC) rule to explicitly introduce the structural congruence. This allows us to specify exactly when the structural congruence has been used, and will later aid our reasoning about the possible evaluations that a given process may perform. The ($\Downarrow$COMP) (composition) rule is fairly obvious, having two processes communicating and then continuing to evaluate to C. The ($\Downarrow$REC) (recursion) rule may seem slightly odd. A possibly more obvious rule may have been (if we had also added an explicit rule for parallel composition):

$$(\Downarrow \text{REC'}) \quad \frac{P[\mathbf{rec}\ X.P/X] \Downarrow C}{\mathbf{rec}\ X.P \Downarrow C}$$

However this does not give the desired or expected behaviour for some processes. For example (remembering note 2.13):

$$a.0|\mathbf{rec}\ X.(\overline{b}.0(b.0 + b.X)) \not\Downarrow a.0$$

One might also have thought that having a rule for summation rather than including it in ($\Downarrow$PRE) would have been a good idea. There are two rules for summation that come to mind (remembering the problems with recursion). We introduce them, temporarily, as examples of what may occur. One of them is

$$(\Downarrow \text{SUM}) \quad \frac{S_1 Q \Downarrow C}{(S_1 + S_2)Q \Downarrow C}$$

This however doesn't give the behaviour of $+$ we desire. We may observe that

$$(a.0 + b.0)c.0 \Downarrow c.a.0$$

which is characteristic of internal choice $\oplus$[1] and can in fact be encoded as

$$S_1 \oplus S_2 \stackrel{\Delta}{=} \nu n.(\overline{n}.0(n.S_1 + n.S_2)) \quad n \text{ fresh}$$

We observe from the following derivation that our encoding does allow the derivation given by ($\Downarrow$SUM), noting that we may assume that n is not free in Q since n is bound:

$$\frac{\dfrac{\overline{n}.0 \Downarrow \overline{n}.0 \quad (n.S_1 + n.S_2)Q \Downarrow n.(S_1 Q) \quad \nu n.(S_1 Q) \Downarrow C'}{\nu n.((\overline{n}.0(S_1 + S_2))Q) \Downarrow C'} \ (\Downarrow \text{COMP})}{(\nu n.(\overline{n}.0(S_1 + S_2)))Q \Downarrow C} \ (\Downarrow \text{STRUC})$$

where if $C = l.P$ then $C' = l.\nu n.P$ and $\nu n.(S_1 Q) \Downarrow C'$ may be derived from $S_1 Q \Downarrow C$.[2]. The characteristic of internal choice is that neither S_1 nor S_2 need reduce in order for the choice to reduce to one or the other. It is easy to see that ($\Downarrow$SUM) defines an internal choice in the manner of [Hen88].

The other version of SUM that we might consider is

$$(\Downarrow \text{SUM}') \quad \frac{S_1 \Downarrow C}{S_1 \boxplus S_2 \Downarrow C}$$

If we also add the following rules for restriction, parallel composition and a simplified base rule, then we get the same evaluation semantics as given in figure 2.2 on page 26.

$$(\Downarrow \text{COM}) \quad \frac{}{l.P \Downarrow l.P}$$

$$(\Downarrow \text{RES}) \quad \frac{P \Downarrow l.P'}{\nu n.P \Downarrow l.\nu n.P'} \ (l, \bar{l} \neq n)$$

$$(\Downarrow \text{PAR}) \quad \frac{P \Downarrow l.P'}{PQ \Downarrow l.(P'Q)}$$

[1]The idea of internal choice has been presented in various papers, including [BB92] and [Hen88].

[2]This is shown formally in lemma 2.14. Note that we don't use anything here in the proof of lemma 2.14!

Any derivation using the rules in figure 2.2 on page 26 can also be derived using the rules (⇓COM), (⇓RES), (⇓REC), (⇓COMP), (⇓SUM′), (⇓PAR) and (⇓STRUC), since (⇓PRE) can be written in terms of (⇓COM), (⇓SUM′), (⇓PAR) and (⇓RES). The converse of this, and hence the fact that we may use the smaller rule set, is given by the following lemma, having noted that all sums are guarded:

Lemma 2.14

If $P \Downarrow l.P'$ and $l, \bar{l} \neq n$, then $\nu n.(PQ) \Downarrow l.\nu n.(P'Q)$.

Proof

This follows by induction on derivation of $P \Downarrow l.P'$. □

We can also show that we can remove name restrictions as well, but first give a lemma about structural congruence that will aid the proofs.

Lemma 2.15

1. If $\nu n.P \equiv P_1 P_2$ then either
   - there exists P_1' with $\nu n.P_1' \equiv P_1$ and $P \equiv P_1'|P_2$.
   - there exists P_2' with $\nu n.P_2' \equiv P_2$ and $P \equiv P_1|P_2'$.
2. If $\nu n.P \equiv \nu \vec{m}_i.P'$ then one of the following must hold
   - there exists a P'' with $\nu n.P'' \equiv P'$ and $P \equiv \nu \vec{m}_i.P''$
   - there exists a j with $m_j = n$, $P \equiv \nu \vec{m}''.P'$ and $\nu n.\nu \vec{m}''.P' \equiv \nu \vec{m}_i.P'$ where $\vec{m}'' = \vec{m}_i \setminus \bigcup_{k:m_k=n} m_k$.
   - there exists an m_j and a fresh name m' such that

$$\begin{aligned} m_i' &= m' && \text{if } m_i = n \\ &\quad m_i && \text{otherwise} \\ \vec{m}'' &= \bigcup_{i \neq j} m_i' \end{aligned}$$

$P \equiv \nu \vec{m}''.P'[n/m_j, m'/n]$ and $\nu n.\nu \vec{m}''.P'[n/m_j, m'/n] \equiv \nu \vec{m}_i.P'$.

Proof

The first part follows easily from the definition of structural congruence. Once it is observed that the definition of $\vec{m}''$ is that it renames n to a fresh name and m_j to n then the second part also follows easily. □

Lemma 2.16

For any NCCS expressions P and P' and any label l with $P \Downarrow l.P'$ then for any name n and expression P_1 with $P \equiv \nu n.P_1$ then there is an expression P_1' with $P_1 \Downarrow l.P_1'$ and $\nu n.P_1' \equiv P'$.

Proof

We proceed by induction on the derivation of $P \Downarrow l.P'$. The last rule used in the derivation was:

$\Downarrow$**PRE** Therefore $P = \nu\vec{m}.((l.P'' + S)R)$ for some $\vec{m}$, P'', S and R. We also see that $P' = \nu\vec{m}.(P''|R)$. The result now follows from lemma 2.15.

$\Downarrow$**COMP** Therefore $P = \nu\vec{m}.(Q_1|Q_2)$. By lemma 2.15 part 2 there are three cases.

- There is a P_2 with $\nu n.P_2 \equiv Q_1|Q_2$ and $P_1 \equiv \nu\vec{m}.P_2$. Then by lemma 2.15 part 1 there are two cases. These cases are essentially the same so we only give the first case, where there is a P_3 with $\nu n.P_3 \equiv Q_1$, n not free in Q_2 and $P_2 \equiv P_3|Q_2$. The second, where there is a P_3 with $\nu n.P_3 \equiv Q_2$, n not free in Q_1 and $P_2 \equiv Q_1|P_3$, follows in exactly the same way. So $Q_1 \Downarrow l'.Q_1'$ and hence by induction hypothesis there is a P_3' with $\nu n.P_3' \equiv Q_1'$ and $P_3 \Downarrow l'.P_3'$. But we know that $Q_2 \Downarrow \overline{l}'.Q_2'$ for some Q_2'. Then, possibly using α-conversion on the names in $\vec{m}$, we get $\nu n.\nu\vec{m}.(P_3'|Q_2') \equiv \nu\vec{m}.(Q_1'|Q_2')$. The result now follows by using the induction hypothesis.

- Hence there are $\vec{m}'$ with $P_1 \equiv \nu\vec{m}'.(Q_1|Q_2)$ and $\nu n\nu\vec{m}'.(Q_1|Q_2) \equiv P$. Then, as above, $Q_1 \Downarrow l'.Q_1'$ and $Q_2 \Downarrow \overline{l}'.Q_2'$ for some Q_1' and Q_2'. But we see that $\nu n.\nu\vec{m}'.(Q_1'|Q_2') \equiv \nu\vec{m}.(Q_1'|Q_2')$ and hence the result follows from the induction hypothesis.

- This case follows in the same way as the previous case.

$\Downarrow$**REC** This follows in essentially the same way as the $\Downarrow$COMP case above.

$\Downarrow$**STRUC** Hence $P \equiv Q$ and $P' \equiv Q'$. Therefore $\nu n.P_1 \equiv Q$. Therefore by induction hypothesis there is a P_1' with $\nu n.P_1' \equiv Q'$ and $P_1 \Downarrow l.P_1'$. Hence $P_1' \equiv P'$ as required.

□

Corollary 2.17

If $\nu n.P \Downarrow l.P'$ then there exists a P'' with $P \Downarrow l.P''$ and $\nu n.P'' \equiv P'$.

Proof

Follows as a special case of lemma 2.16. □

Using the evaluation rules given in figure 2.2 on page 26 we are implicitly restricted to only permitting guarded summation in the calculus. However, by using the alternative characterisation of evaluation given above, we are not so restricted. One might then ask why we restrict the syntax to only allowing guarded summation. There are two principal reasons for accepting this restriction.The first is that the tie up, given in section 6 of this chapter, and also in chapter 5, between the "big step" evaluation relation and the "small step" transition relation, is closer with this restriction. The second is that the equivalence relations given in chapter 3 are all congruences under this restriction, while some, for example the weak $\Downarrow$-bisimilarity given in definition 3.8 on page 44, are not congruences in the presence of unguarded summation. This is the principal reason why we have adopted guarded summation for use in NCCS.

2.4 Syntax of τNCCS

We define τNCCS similarly to NCCS, changing the definitions of S and G to give the syntax of τNCCS processes as given in figure 2.4 on the next page.

Using Σ-notation for summation may not seem natural. The reason for using this form of summation, rather than the binary summation used in NCCS, is that it simplifies both the translation of τNCCS into NCCS, and the equational theory for NCCS given in chapter 7. The removal of the null process from the list of possible values for S is a slight of hand, since we may retrieve it simply by having an empty sum. For convenience we may split up

P	::=	X	variable
	|	PP	binary composition
	|	S	summation
	|	$\nu n.P$	restriction
	|	rec $X.P$	recursion
S	::=	$\sum_{i\in I} G_i$ (I finite)	finite summation
G	::=	$n.P$	input guarded process
	|	$\overline{n}.P$	output guarded process
	|	$\tau.P$	τ guarded process

Figure 2.4: Syntax of τNCCS

sums thus:

$$\sum_{i\in I\uplus J} P_i = \sum_{i\in I} P_i + \sum_{j\in J} P_j$$

By using this notation, we implicitly have associativity and commutativity of summation. It also allows sums that include the null process 0, having noted that:

$$\sum_{I} P_i = \sum_{I} P_i + \sum_{\emptyset} P_j = \sum_{I} P_i + 0$$

One might wonder whether we do want to enforce associativity and commutativity of summation. Certainly a semantics which does not have associativity and commutativity of summation would be considered dubious, but we might prefer these properties to be derivable. We could give the syntax of τNCCS in terms of binary summation and then replace the reduction rule for summation ($(\rightarrow$SUM) in figure 2.5 on page 34) with two new rules:

$$(\rightarrow \text{SUM}_1) \quad \frac{P_1 \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P}$$

$$(\rightarrow \text{SUM}_2) \quad \frac{P_2 \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P}$$

Then for any reasonable idea of equivalence, these two versions are equivalent. So we could write (or encode) the syntax and reduction relation in terms of binary sums and derive the desired properties.

Although we have associativity and commutativity of summation enforced implicitly, we do not have a structural congruence relation to enforce other properties.

Definition 2.18

We extend definition 2.3 to τNCCS by adding the following additional rule and repeatedly splitting up sums until they are binary sums.

$$\text{fn}(\tau.P) \quad = \quad \text{fn}(P)$$

We observe that τNCCS is essentially the syntax used for CCS[Mil89] with the restriction to guarded summation.

2.5 Transition relation for τNCCS

The evaluation semantics for τNCCS are specified by the inductively defined "small step" labelled transition relation given in figure 2.5 on the next page.

Note 2.19

The reduction relation given in figure 2.5 is the same as the reduction relation for CCS, once the syntax has been translated.

$$(\rightarrow \text{ACT}) \qquad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$(\rightarrow \text{SUM}) \qquad \frac{G_i \xrightarrow{\alpha} P}{\sum_{i \in I} G_i \xrightarrow{\alpha} P}$$

$$(\rightarrow \text{COMM}_1) \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 P_2 \xrightarrow{\alpha} P_1' P_2}$$

$$(\rightarrow \text{COMM}_2) \qquad \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 P_2 \xrightarrow{\alpha} P_1 P_2'}$$

$$(\rightarrow \text{COMM}_3) \qquad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{PQ \xrightarrow{\tau} P'Q'}$$

$$(\rightarrow \text{RES}) \qquad \frac{P \xrightarrow{\alpha} P'}{\nu n.P \xrightarrow{\alpha} \nu n.P'} \qquad (n \notin \{\alpha, \overline{\alpha}\})$$

$$(\rightarrow \text{REC}) \qquad \frac{P[\mathbf{rec}\ X.P/X] \xrightarrow{\alpha} P'}{\mathbf{rec}\ X.P \xrightarrow{\alpha} P'}$$

Figure 2.5: Transition relation for τNCCS

2.6 Translations between NCCS and τNCCS

Firstly we note that NCCS is contained in τNCCS by an essentially trivial translation. All we have to do is flatten out the binary sums into a single finite sum, noting that the null process, 0, is just encoded as an empty sum.

Definition 2.20

Given an NCCS process P, $(P)^T$ is the τNCCS process obtained by expanding out nested binary sums into a single flat sum.

Note 2.21

In general we don't write $(-)^T$ explicitly; we just assume it whenever we have an NCCS process and want a τNCCS one.

We do however need a translation, $(-)^t$, to convert τNCCS into NCCS. We obtain this by extending the well-known encoding of $\tau.P$ as $\nu n.(\overline{n}.0|n.P)$, where n is a fresh name.

Definition 2.22

$(-)^t$ is inductively defined by:

$$\begin{array}{rcll}
X^t & \triangleq & X & \\
(\nu m.P)^t & \triangleq & \nu m.(P)^t & \\
(P_1|P_2)^t & \triangleq & (P_1)^t|(P_2)^t & \\
(\mathbf{rec}\ X.P)^t & \triangleq & \mathbf{rec}\ X.(P)^t & \\
(\sum_i \alpha_i.P_i)^t & \triangleq & (\ldots(\alpha_1.(P_1)^t + \ldots) + \alpha_n.(P_n)^t) & \tau \notin \{\alpha_i\} \\
(\sum_i \alpha_i.P_i)^t & \triangleq & (\nu n.(\overline{n}.0|\sum_i \alpha_i'.P_i))^t & \tau \in \{\alpha_i\}
\end{array}$$

where

$$\begin{array}{rcll}
\alpha_i' & = & n & \alpha_i = \tau \\
 & = & \alpha_i & \alpha_i \neq \tau
\end{array}$$

and n is a fresh name.

Note 2.23

We observe that the same fresh name, n above, may be used throughout. It does not stay fresh, but it never becomes a free name and hence it can be renamed using α-conversion. Also note that since it is fresh in the original process expression, the restrictions will never bind a name that is free in the original expression.

Since we do not turn $\nu n.(\overline{n}.0|n.P)$ into $\tau.P$ where appropriate in $(-)^T$, we do not attain a perfect match between NCCS and τNCCS[3]. Also having a structural congruence in NCCS and not in τNCCS will cause other possible mismatches.

Before turning to the relationships we do have we need to recall some properties of CCS strong equivalence that will carry over into τNCCS. We formally define CCS strong equivalence for τNCCS, calling it $\sim_{\rightarrow}$, in definition 3.15 on page 47, but for now we will limit use to the properties proved in chapter 4 of [Mil89]. We will use the following properties of CCS strong equivalence:

- It is a congruence relation [from section 4.4].
- The static laws show that each of the relationships in the structural congruence for NCCS holds for $\sim_{\rightarrow}$. [from Proposition 8]
- The expansion law holds. [from Proposition 9]

The expansion law, together with the static laws, will be used particularly to show that:

$$\nu n.(\overline{n}.0|n.P) \sim_{\rightarrow} \tau.P \quad n \notin \text{fn}(P)$$

and also for the generalized version for sums.

We can now state some useful relationships we have:

Lemma 2.24

1. If $P \in$ NCCS, then $P^{T^t} \equiv P$
2. If $Q \in \tau$NCCS, then $Q \sim_{\rightarrow} Q^{tT}$
3. If $P \equiv P'$ in NCCS, then $P^T \sim_{\rightarrow} P'^T$ in τNCCS.

Proof

(1) follows naturally from the definition, noting that any sums may have been reordered. (2) uses the expansion law and (3) uses the static laws. □

[3]We could generalize $(-)^T$ to give many possible results and then claim a better match than we currently have. However we gain very little since we still would not get a perfect match.

Corollary 2.25

For any τNCCS process P, there is an NCCS process Q such that:

$$P \sim_{\rightarrow} (Q)^T$$

Proof

We let $Q = (P)^t$. □

Lemma 2.26

For any τNCCS process Q and any open τNCCS process P with X free we have:

$$(P[Q/X])^t \equiv (P)^t[(Q)^t/X]$$

Proof

This follows by induction on the structure of the process expression P, again noting that the order of sums may be different between the two translations.

□

Theorem 2.27

If $\mathcal{R}$ is an equivalence relation on NCCS for which $P \equiv Q$ implies that $P\mathcal{R}Q$, then for any congruence relation, $\mathcal{S}$, on τNCCS

$$P\mathcal{S}Q \Rightarrow P^t\mathcal{R}Q^t$$

and

$$P\mathcal{R}Q \Rightarrow P^T\mathcal{S}Q^T$$

implies that $\mathcal{R}$ is a congruence relation.

Proof

For any (possibly open) NCCS processes P, Q and $C[-]$ with $P\mathcal{R}Q$ then $(C[P])^T = C^T[P^T]$. Similarly $(C[Q])^T = C^T[Q^T]$. Therefore $(C[P])^T\mathcal{S}(C[Q])^T$ since $\mathcal{S}$ is a congruence and $P^T\mathcal{S}Q^T$. Hence we have $(C[P])^{T^t}\mathcal{R}(C[Q])^{T^t}$. But $(C[P])^{T^t} \equiv C[P]$ and $\mathcal{R}$ is an equivalence relation, so by transitivity (twice) and symmetry of $\mathcal{R}$ we have $C[P]\mathcal{R}C[Q]$. Hence $\mathcal{R}$ is a congruence relation.

□

Chapter 3

Bisimulations

We now turn our attention to the notion of equivalence on NCCS and τNCCS processes. We base our equivalences upon the intuitive idea that processes should only be distinguishable by the communications with which they may interact with the environment. In particular we do not wish to observe silent action. This does however present some problems. It may in fact lead to problems with our equivalences not being congruences. We will need to refine this concept since the following two processes are indistinguishable under this intuitive idea:

$$a.0|\overline{a}.0$$

$$a.\overline{a}.0 + \overline{a}.a.0$$

Both these processes can either input or output on channel a and become $\overline{a}.0$ or $a.0$ respectively. However we observe that $a.0|\overline{a}.0|b.0 \Downarrow b.0$ because of the following (example) derivation tree:

$$\dfrac{\dfrac{}{a.0 \Downarrow a.0} \quad \dfrac{\dfrac{\dfrac{}{\overline{a}.0 \Downarrow \overline{a}.0}}{\overline{a}.0|b.0 \Downarrow \overline{a}.(0|b.0)} \quad \dfrac{\dfrac{}{b.0 \Downarrow b.0}}{0|b.0 \Downarrow b.(0|0)}}{\overline{a}.0|b.0 \Downarrow \overline{a}.b.0} \quad \dfrac{}{0|b.0 \Downarrow b.0}}{a.0|\overline{a}.0|b.0 \Downarrow b.0}$$

(using ($\Downarrow$PRE) three times, ($\Downarrow$STRUC) twice and ($\Downarrow$COMP) once)

But $(a.\overline{a}.0 + \overline{a}.a.0)|b.0$ cannot evaluate in one step to $b.0$. We may also observe that the above two processes have identical traces. Branching time congruences have been useful in practice for CCS-like calculi. We present a modification of this method for NCCS involving parallel contexts

of varying degrees of freedom. We also give the relationship between these equivalences and various others based on CCS-like languages.

In this chapter we summarise the main results, giving forward references to the proofs. The two main results that are not included are the relationship between evaluation and transition, given in chapter 5, and the equational characterisation of evaluation bisimulation (see definition 3.4 on page 42) which we give in chapter 7. As a basic requirement of our bisimulations we desire that two bisimilar processes may be exchanged within a program such that the two versions of the programs cannot be distinguished by the corresponding bisimulation, i.e. we desire that our bisimulations are congruences. Before defining the bisimulations we present various definitions.

Definition 3.1

We extend a binary relation $\mathcal{R} \subseteq$ closed (τ)NCCS $\times$ closed (τ)NCCS to a binary relation $\mathcal{R}^o \subseteq (\tau)$NCCS $\times$ (τ)NCCS by substitution of closed processes for free variables. That is for any processes P, Q if $P\mathcal{R}^oQ$ then for $X_i \in \mathrm{fv}(P) \uplus \mathrm{fv}(Q)$

$$\forall R_i \in \text{closed } (\tau)\text{NCCS } \; P[R_i/X_i]\mathcal{R}Q[R_i/X_i]$$

We call $\mathcal{R}^o$ the extension of $\mathcal{R}$ to open processes.

Definition 3.2

A binary relation $\mathcal{R} \subseteq (\tau)$NCCS $\times$ (τ)NCCS is a *congruence* if it is an equivalence relation and for any (τ)NCCS processes P, Q and an open (τ)NCCS process C with at least one free variable X

$$P\mathcal{R}Q \Rightarrow \; C[P/X]\mathcal{R}C[Q/X]$$

The substitution of P/Q for X allows for free names and variables in P/Q, respectively, to be bound by C.

The presence of parallel composition and communication introduces non-determinacy into (τ)NCCS. This means that we are led to take a number of decisions about what provides a useful equivalence. The first is whether we are to regard two equivalent processes as behaving the same way in terms of

must or *may*. Having non-determinacy, there are processes which will not be equivalent to themselves in must terms. For example the following process:

$$\nu a.(\overline{a}.0|a.\overline{b}.0|a.\overline{c}.0)$$

which may either output b or c and then become the null (or dead) process. As a consequence of this we construct our equivalences in may terms. The second of these decisions concerns whether we are interested in one sided simulations, constructing our equivalences in terms of simulations, or whether we needs to start with bisimulations. Again the choice here is a consequence of non-determinacy. We may have two processes such that each can simulate the other, but which are not bisimilar![1] For example

$$P = a.b.0 + a.\nu n.(\overline{n}.0|(n.b.0 + n.c.0))$$
$$Q = a.c.0 + a.\nu n.(\overline{n}.0|(n.b.0 + n.c.0))$$

can both simulate each other if we take our simulation to be half the bisimulation defined in definition 3.4 overleaf. However if $P \Downarrow a.b.0$ then we must have $Q \Downarrow a.\nu n.(\overline{n}.0|(n.b.0 + n.c.0))$ for Q to simulate P, but $\nu n.(\overline{n}.0|(n.b.0 + n.c.0)) \Downarrow c.0$ which $b.0$ cannot do. Therefore, in order to get evaluation based characterisations of branching time congruences, we construct our equivalences in terms of bisimulations.

Definition 3.3

For a function $\mathcal{F}$, over binary relations, $\mathcal{R}$, on (τ)NCCS, such that $\mathcal{F}$ is monotonic, that is $\mathcal{R}_1 \subseteq \mathcal{R}_2$ implies $\mathcal{F}(\mathcal{R}_1) \subseteq \mathcal{F}(\mathcal{R}_2)$, then

- $\mathcal{R}$ is a *fixed point* if $\mathcal{R} = \mathcal{F}(\mathcal{R})$.
- $\mathcal{R}$ is a *post-fixed point* if $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$.
- $\nu\mathcal{R}.\mathcal{F}(\mathcal{R})$ is the *greatest post-fixed point.* This will be a fixed-point since $\mathcal{F}$ is monotonic. This exists by Tarski's fixed point theorem [Tar55].

[1] This has been observed by among others Milner. See Exercise 14 in [Mil89].

3.1 Evaluation Bisimilarity

Having taken the premise that the basic operation in process calculi is parallel composition, the natural way to define a notion of equivalence is by allowing testing in the presence of a parallel context. The following definition gives our basic bisimulation on NCCS, which we go on to prove is a congruence in chapter 4.

Definition 3.4 *Evaluation Bisimulation*

Given $\mathcal{R} \subseteq \text{NCCS} \times \text{NCCS}$, define

$$\begin{aligned} P\{\mathcal{R}\}Q \triangleq\ & \forall R, l, P' \ \ PR \Downarrow l.P' \Rightarrow \\ & \qquad \exists Q' \ (QR \Downarrow l.Q' \ \ \& \ \ P'\mathcal{R}Q') \\ \&\ & \forall R, l, Q' \ \ QR \Downarrow l.Q' \Rightarrow \\ & \qquad \exists P' \ (PR \Downarrow l.P' \ \ \& \ \ P'\mathcal{R}Q') \end{aligned}$$

Let $\simeq_{\Downarrow}$ be $\nu\mathcal{R}.\{\mathcal{R}\}$, and $\simeq^o$ be the extension to open processes. We shall call $\simeq_{\Downarrow}$ evaluation bisimulation.

Note 3.5

We often write $\simeq_{\Downarrow}$ as $\simeq$ where the context allows.

Working with unrestricted contexts gives rise to an equivalence that has all the desired properties, but is extremely difficult to work with. We can however restrict the contexts and we prove in chapter 6 that $\simeq'_{\Downarrow}$, defined below, is equivalent to $\simeq_{\Downarrow}$, defined in definition 3.4.

Definition 3.6 *Restricted Evaluation Bisimulation*

Given $\mathcal{R} \subseteq \text{NCCS} \times \text{NCCS}$, define

$$\begin{aligned} P\{\mathcal{R}\}_r Q \triangleq\ & \forall x \notin \text{fn}(P,Q), l, P' \ \ P|x.0 \Downarrow l.P' \Rightarrow \\ & \qquad \exists Q' \ (Q|x.0 \Downarrow l.Q' \ \& \ P'\mathcal{R}Q') \\ \&\ & \forall x \notin \text{fn}(P,Q), l, Q' \ \ Q|x.0 \Downarrow l.Q' \Rightarrow \\ & \qquad \exists P' \ (P|x.0 \Downarrow l.P' \ \& \ P'\mathcal{R}Q') \end{aligned}$$

Then let $\simeq'_{\Downarrow}$ be $\nu\mathcal{R}.\{\mathcal{R}\}_r$.

Theorem 3.7

1. $\simeq^o_{\Downarrow}$ is an equivalence relation.
2. $\simeq^o_{\Downarrow}$ is a congruence relation.
3. $\simeq_{\Downarrow}$ coincides with $\simeq'_{\Downarrow}$

Proof

1. The identity relation is certainly reflexive and is also a post-fixed point since any process can do the same as itself. Therefore the greatest post-fixed point is reflexive.

   If $\mathcal{R}$ is a fixed point of $\{-\}_r$ then for any processes P, Q, R with $P\mathcal{R}Q$ and $Q\mathcal{R}R$ we can deduce that $P\mathcal{R}R$ by "chaining together" the relations. We do this using the same context, and substitutions if the processes are open. This will mean that we may deduce an evaluation for R from an evaluation for P via an evaluation for Q. Therefore $\mathcal{R}$ is transitive.

   Symmetry follows by observing that if $\mathcal{R}$ is a fixed point then $\mathcal{R}^{-1}$ is also. Therefore $\mathcal{R} \cup \mathcal{R}^{-1}$ is also a fixed point. Since $\simeq^o_{\Downarrow}$ is the greatest post-fixed point it must be equal to $\simeq^o_{\Downarrow} \cup (\simeq^o_{\Downarrow})^{-1}$ which is symmetric.

2. We prove this in chapter 4.
3. We prove this, in passing, in section 6.4.

□

3.2 Weak ⇓-bisimilarity

We also provide various bisimulations that relate to well known equivalences in other calculi. The first we give here is weak bisimilarity defined for NCCS. We will prove that it is the same as CCS weak equivalence. However, in NCCS, summation is restricted to label guarded summation so it will be a congruence.

Definition 3.8 *Weak Bisimulation*

Given $\mathcal{R} \subseteq \text{NCCS} \times \text{NCCS}$ define $\{\mathcal{R}\}_w \subseteq \text{NCCS} \times \text{NCCS}$ by:

$$\begin{aligned} P\{\mathcal{R}\}_w Q \;\triangleq\; & \forall l, P' \; \forall n \notin \text{fn}(l, P, Q, P') \;\; P|l.n.0 \Downarrow n.P' \Rightarrow \\ & \exists Q' \; (Q|l.n.0 \Downarrow n.Q' \;\; \& \;\; P'\mathcal{R}Q') \\ \& \;\; & \forall l, Q' \; \forall n \notin \text{fn}(l, P, Q, Q') \;\; Q|l.n.0 \Downarrow n.Q' \Rightarrow \\ & \exists P' \; (P|l.n.0 \Downarrow n.P' \;\; \& \;\; P'\mathcal{R}Q') \\ \& \;\; & \forall P' \; \forall n \notin \text{fn}(P, Q, P') \;\; P|n.0 \Downarrow n.P' \Rightarrow \\ & \exists Q' \; (Q|n.0 \Downarrow n.Q' \;\; \& \;\; P'\mathcal{R}Q') \\ \& \;\; & \forall Q' \; \forall n \notin \text{fn}(P, Q, Q') \;\; Q|n.0 \Downarrow n.Q' \Rightarrow \\ & \exists P' \; (P|n.0 \Downarrow n.P' \;\; \& \;\; P'\mathcal{R}Q') \end{aligned}$$

Let $\approx_{\Downarrow}$ be $\nu\mathcal{R}.\{\mathcal{R}\}_w$.

Note 3.9

We will often write $\approx_{\Downarrow}$ as $\approx$.

Theorem 3.10

$\approx_{\Downarrow}$ is a congruence relation.

Proof

$\approx_{\Downarrow}$ is an equivalence relation, with the proof following in the same way as theorem 3.7(1). We prove congruence in section 6.2. □

Theorem 3.11

For NCCS processes P and Q $\;P \simeq_{\Downarrow} Q$ implies $P \approx_{\Downarrow} Q$.

Proof

For $P \simeq Q$

- If $P|l.n.0 \Downarrow n.P'$ then there is a Q' with $Q|l.n.0 \Downarrow n.Q'$ and $P' \simeq Q'$ since $P \simeq Q$.
- If $P|n.0 \Downarrow n.P'$ then there is a Q' with $Q|n.0 \Downarrow n.Q'$ and $P' \simeq Q'$ since $P \simeq Q$.

$\simeq$ is a post-fixed point of $\{-\}_w$ by a symmetric argument. Therefore $P \simeq Q$ implies $P \approx Q$ as required (noting results in definition 3.3 on page 41). □

We may wonder whether the inclusion $\simeq \subseteq \approx$ is strict. It is, as shown by the following pair of processes:

$$P = a.(\nu n.(\overline{n}.0|(b.0 + n.c.0)))$$
$$Q = a.(\nu n.(\overline{n}.0|(b.0 + n.c.0))) + a.c.0$$

These may be clearer when written in τNCCS where they are:

$$P' = a.(b.0 + \tau.c.0)$$
$$Q' = a.(b.0 + \tau.c.0) + a.c.0$$

Then we note that $P \not\simeq Q$ but $P \approx Q$.[2]

3.3 Barbed $\Downarrow$-bisimilarity

The next bisimulation we give is a weak version of the notion of the barbed bisimulation of Milner and Sangiorgi, described in [MS92]. The barbed bisimulation is an attempt to provide an equivalence that can be used in various calculi, both lower and higher order, and hence enable comparisons between calculi. A consequence of the evaluation relation being a "big step" relation is that we cannot define the strong barbed bisimulation. Two methods of extending barbed bisimulation to the weak case are given in [MS92]. One is to increase the power of the observer by partitioning the visible actions into subsets and then allowing the observer to discriminate between visible actions from different subsets. This approach is followed in [San92] in which he shows that his weak barbed bisimulation is the same as weak CCS equivalence. The other approach is to allow a set of possible contexts for which the relation holds, with one equivalence for each different set of allowable contexts. This approach leads to an equivalence for which it is harder to show that two processes are equivalent, since we have to do this over a possibly infinite set of contexts. We will however follow this approach and show that we can restrict the contexts to some very simple cases. We shall also prove that weak barbed bisimulation is the same as weak bisimulation for NCCS, which in turn is the same as weak CCS equivalence (theorem 3.21 on page 48).

[2] This is just one of the τ laws in [Mil89].

Definition 3.12 *Barbed Bisimulation*

Given $\mathcal{R} \subseteq \text{NCCS} \times \text{NCCS}$ define $\{\mathcal{R}\}_b \subseteq \text{NCCS} \times \text{NCCS}$ by:

$$\begin{aligned} P\{\mathcal{R}\}_b Q \quad \triangleq \quad & \forall C[-] \;\; C[P] \Downarrow \Rightarrow C[Q] \Downarrow \\ \& \quad & \forall C[-] \;\; C[Q] \Downarrow \Rightarrow C[P] \Downarrow \\ \& \quad & \forall C[-], P' \; \forall n \notin \text{fn}(P, P', Q, C) \;\; C[P] | n.0 \Downarrow n.P' \Rightarrow \\ & \quad \exists Q' \; (C[Q] | n.0 \Downarrow n.Q' \;\; \& \;\; P' \mathcal{R} Q') \\ \& \quad & \forall C[-], Q' \; \forall n \notin \text{fn}(P, Q, Q', C) \;\; C[Q] | n.0 \Downarrow n.Q' \Rightarrow \\ & \quad \exists P' \; (C[P] | n.0 \Downarrow n.P' \;\; \& \;\; P' \mathcal{R} Q') \end{aligned}$$

where $P \Downarrow$ means $\exists l, P'$ s.t. $P \Downarrow l.P'$ and $C[-]$ is any context.
Let $\approx^b_{\Downarrow}$ be $\nu\mathcal{R}.\{\mathcal{R}\}_b$.

Note 3.13

We will often write $\approx^b_{\Downarrow}$ as $\approx$.

Theorem 3.14

1. $\approx^b_{\Downarrow}$ is a congruence relation and coincides with $\approx_{\Downarrow}$.
2. If $C[-]$ is restricted to be of the form $\nu\vec{n}.(-|l_1.l_2.0|\overline{l_2}.l_3.0)$, where l_2, l_3 are fresh names, then the relation remains the same.

Proof

We prove both parts in section 6.3. □

3.4 Strong $\rightarrow$-bisimilarity

We turn our attention to τNCCS. In section 3.6, we give a version of Weijland's delay bisimulation from his PhD thesis [Wei89]. We first define the strong and weak equivalences from CCS. The operational semantics of CCS and τNCCS being essentially[3] the same mean that we may just give the equivalences and use the results already known about them. Firstly we define Strong $\rightarrow$-bisimilarity, also known as strong equivalence in CCS[Mil89].

[3] As discussed in section 2.4 on page 31 the summation notation may be interchanged with binary summation.

Definition 3.15 *Strong CCS Equivalence*

Given $\mathcal{S} \subseteq \tau\text{NCCS} \times \tau\text{NCCS}$, define $[\mathcal{S}] \subseteq \tau\text{NCCS} \times \tau\text{NCCS}$ by:

$$\begin{aligned} P[\mathcal{S}]Q \quad \hat{\Leftrightarrow} \quad & \forall \alpha, P' \ (P \xrightarrow{\alpha} P' \Rightarrow \exists Q' \ (Q \xrightarrow{\alpha} Q' \ \& \ P' \mathcal{S} Q')) \\ \& \quad & \forall \alpha, Q' \ (Q \xrightarrow{\alpha} Q' \Rightarrow \exists P' \ (P \xrightarrow{\alpha} P' \ \& \ P' \mathcal{S} Q')) \end{aligned}$$

Let $\sim_{\rightarrow}$ be $\nu\mathcal{S}.[\mathcal{S}]$.

Note 3.16

We normally write $\sim$ instead of $\sim_{\rightarrow}$ where the context allows it to be clear.

Theorem 3.17

$\sim_{\rightarrow}$ is a congruence relation.

Proof

This is just the strong congruence of CCS[Mil89]. □

Lemma 3.18

For any τNCCS processes P and Q, $P \sim Q$ implies $P^t \simeq_{\Downarrow} Q^t$

Proof

This is one part of lemma 6.1. □

One may ask whether the converse of theorem 3.18 is also true. It is not since $(a.0)^t \simeq_{\Downarrow} (\tau.a.0)^t$ but $a.0 \not\sim \tau.a.0$.

3.5 Weak $\rightarrow$-bisimilarity

We now define $\approx_{\rightarrow}$ which is also known as weak equivalence in CCS.

Definition 3.19 *Weak CCS Equivalence*

Given $\mathcal{S} \subseteq \tau\text{NCCS} \times \tau\text{NCCS}$ define $[\mathcal{S}]_w \subseteq \tau\text{NCCS} \times \tau\text{NCCS}$ by:

$$\begin{aligned} P[\mathcal{S}]_w Q \quad \hat{\Leftrightarrow} \quad & \forall l, P' \ P \xrightarrow{l} P' \Rightarrow \exists Q' \ Q \xrightarrow{\tau^* l \tau^*} Q' \ \& \ P' \mathcal{S} Q') \\ \& \quad & \forall l, Q' \ Q \xrightarrow{l} Q' \Rightarrow \exists P' \ P \xrightarrow{\tau^* l \tau^*} P' \ \& \ P' \mathcal{S} Q') \\ \& \quad & \forall P' \ P \xrightarrow{\tau} P' \Rightarrow \exists Q' \ (Q \xrightarrow{\tau^*} Q' \ \& \ P' \mathcal{S} Q') \\ \& \quad & \forall Q' \ Q \xrightarrow{\tau} Q' \Rightarrow \exists P' \ (P \xrightarrow{\tau^*} P' \ \& \ P' \mathcal{S} Q') \end{aligned}$$

Let $\approx_{\rightarrow}$ be $\nu\mathcal{S}.[\mathcal{S}]_w$.

Note 3.20

We will write $\approx_{\rightarrow}$ as $\approx$ where the context allows.

Theorem 3.21

1. $\approx_{\rightarrow}$ is a congruence relation,

2. For NCCS processes P and Q and τNCCS processes R and S then $P \approx_{\Downarrow} Q$ implies that $P^T \approx_{\rightarrow} Q^T$ and $R \approx_{\rightarrow} S$ implies that $R^t \approx_{\Downarrow} S^t$. Thus $\approx_{\Downarrow}$ and $\approx_{\rightarrow}$ coincide under translation.

Proof

1. $\approx_{\rightarrow}$ is just the CCS weak equivalence which is a congruence when summation is restricted to guarded summation. See [Mil89] for details.

2. This is given in section 6.2.

□

3.6 Delay $\rightarrow$-bisimilarity

Weijland introduces the idea of *delay bisimulation* in his thesis [Wei89][4]. Translating between notations, his delay bisimulation excludes the following relation:

$$a.(\tau.P + Q) = a.(\tau.P + Q) + a.P$$

where we assume that Q is a sum. This is a valid relationship in weak bisimulation. It is not in delay bisimulation because we "take a snapshot" of the processes as soon as the input or output happens. Hence the right hand side cannot input on channel a and then become P; it must become $\tau.P + Q$.

We give here the equivalent notion for τNCCS and will go on to prove that it is the same as $\simeq_{\Downarrow}$.

[4]Note that in his thesis he uses ax to mean $a.x$ and $a \parallel b$ where we use ab or $a|b$. Also he allows processes to be prefixes of processes, removing the guard once they have finished executing.

Definition 3.22 *Delay Bisimulation*

Given $\mathcal{S} \subseteq \tau\text{NCCS} \times \tau\text{NCCS}$ define $[\mathcal{S}]_d \subseteq \tau\text{NCCS} \times \tau\text{NCCS}$ by:

$$\begin{aligned} P[\mathcal{S}]_d Q \quad \triangleq \quad & \forall l, P'\ P \xrightarrow{l} P' \Rightarrow \exists Q'\ Q \xrightarrow{\tau^* l} Q' \ \ \& \ \ P'\mathcal{S}Q') \\ \& \quad & \forall l, Q'\ Q \xrightarrow{l} Q' \Rightarrow \exists P'\ P \xrightarrow{\tau^* l} P' \ \ \& \ \ P'\mathcal{S}Q') \\ \& \quad & \forall P'\ P \xrightarrow{\tau} P' \Rightarrow \exists Q'\ (Q \xrightarrow{\tau^*} Q' \ \ \& \ \ P'\mathcal{S}Q') \\ \& \quad & \forall Q'\ Q \xrightarrow{\tau} Q' \Rightarrow \exists P'\ (P \xrightarrow{\tau^*} P' \ \ \& \ \ P'\mathcal{S}Q') \end{aligned}$$

Let $\simeq^d_{\rightarrow}$ be $\nu\mathcal{S}.[\mathcal{S}]_d$.

Note 3.23

As usual we will often write $\simeq^d_{\rightarrow}$ as just $\simeq$ where the context allows. Also the resulting equivalence differs from Weijland's delay bisimulation due to differences in syntax between τNCCS and his calculus.

Theorem 3.24

1. $\simeq^d_{\rightarrow}$ is a congruence relation.
2. $\simeq^d_{\rightarrow}$ coincides with $\simeq_{\Downarrow}$

Proof

The proofs for both parts are given in section 6.4. □

To summarise the relationship between the equivalences we have the following chain of inclusions:

$$\sim \subsetneq \simeq_{\Downarrow} \equiv \simeq'_{\Downarrow} \equiv \simeq^d_{\rightarrow} \subsetneq \approx_{\Downarrow} \equiv \approx^b_{\Downarrow} \equiv \approx_{\rightarrow}$$

Hence we can see that it is reasonable to refer to the equivalences merely as $\sim$, $\simeq$ or $\approx$ as appropriate.

Chapter 4

Proof of congruence of $\simeq^{O}_{\Downarrow}$

We will proceed using a modified version of Howe's method [How89], but also see [How96]. One might wonder why we use such a method rather than a direct proof. The main reason is that for any evaluation $P \Downarrow C$, C can be structurally very different from P. This is sharply different from the case involving a labelled transition relation where for $P \xrightarrow{\alpha} P'$, P and P' are structurally similar. This difference in structure would make a direct proof extremely difficult.

The method we use is based upon defining an auxiliary relation which is easier to work with and then proving that it is equivalent to the original relation. In this case, due to the non-determinism and the use of a structural congruence in NCCS, the proof is longer and more complex than in [How89]. It proceeds in this case in the following manner:

1. We define an auxiliary relation ($\simeq^*$). We do this in section 4.1 as well as proving various lemmas which will be useful later in the proof.

2. We show that $\simeq^{o}_{\Downarrow}$ is included in $\simeq^*$ and that $\simeq^*$ has a substitutivity property and that it is reflexive. This is done in section 4.2.

3. $\simeq^*$ is then shown to be preserved under the evaluation relation. This occurs in section 4.3.

4. We then prove that $\simeq^*$ is symmetric under transitive closure. We do this rather than showing directly that it is symmetric because it is easier

and doesn't increase the complexity of the rest of the proof significantly. This comes in section 4.4.

5. Finally we can prove the main result that $\simeq^*$ and $\simeq^o$ coincide and hence that $\simeq^o$ is a congruence. This is in section 4.5.

4.1 Auxiliary relation

We now define the Auxiliary relation $E \simeq^* F$. It is defined in terms of $\simeq^o$ such that we can deduce the last rule used in the derivation of $E \simeq^* F$, apart from (STRUC), from the structure of E. This allows us to prove various results more easily since we have a stronger grasp on the relation between the two processes. To overcome the problems of having a structural congruence relation we include the (STRUC) rule. This however causes problems with structural induction proofs which rely on knowing the last rule used in the derivation of $E \simeq^* F$. We remove this difficulty by showing that we may restrict derivations to having the (STRUC) rule used at most once, and then as the last rule of the derivation. As a consequence of this we generally prove results in two stages. First we prove the result for $E \simeq^* F$ whose derivation does not use (STRUC) at all, and then extend the result to the general case.

Definition 4.1

We inductively define the auxiliary relation $\simeq^*$ using the rules given in figure 4.1 on the next page.

4.2 Initial Lemmas

We start by examining the restrictions we may place on the required derivations of $E \simeq^* F$. We first define two restrictions on derivations and then prove that all derivations are equivalent to one or the other.

Definition 4.2

A derivation of $E \simeq^* F$ is *normal* if the (STRUC) rule is used at most once, and then as the last rule of the derivation. Furthermore, it is *simple* if it does not use the (STRUC) rule at all. We say that $E \simeq^* F$ is normal/simple if the derivation we will use is normal/simple.

(NULL)	$\dfrac{}{0 \simeq^* F}$	if $0 \simeq^o_{\Downarrow} F$
(VAR)	$\dfrac{}{X \simeq^* F}$	if $X \simeq^o_{\Downarrow} F$
(COM)	$\dfrac{E \simeq^* E'}{l.E \simeq^* F}$	if $l.E' \simeq^o_{\Downarrow} F$
(REC)	$\dfrac{E \simeq^* E'}{\mathrm{rec}X.E \simeq^* F}$	if $\mathrm{rec}X.E' \simeq^o_{\Downarrow} F$
(COMP)	$\dfrac{E_1 \simeq^* E'_1 \quad E_2 \simeq^* E'_2}{E_1 E_2 \simeq^* F}$	if $E'_1 E'_2 \simeq^o_{\Downarrow} F$
(SUM)	$\dfrac{E_1 \simeq^* E'_1 \quad E_2 \simeq^* E'_2}{E_1 + E_2 \simeq^* F}$	if $E'_1 + E'_2 \simeq^o_{\Downarrow} F$
(RES)	$\dfrac{E \simeq^* E'}{\nu n.E \simeq^* F}$	if $\nu n.E' \simeq^o_{\Downarrow} F$
(STRUC)	$\dfrac{E' \simeq^* F}{E \simeq^* F}$	if $E \equiv E'$

Figure 4.1: Auxiliary relation

Lemma 4.3

If $E \simeq^* F$ then there exists a normal derivation for it.

Proof

The (STRUC) rule may be "pushed down" through all the other rules since $\equiv$ is a congruence. Also, since $\equiv$ is transitive, two sequential (STRUC) rules may be unified into one. □

We now justify that the auxiliary relation contains Evaluation Bisimulation. This is the first criterion for the auxiliary relation to be useful. The other parts are that the auxiliary relation has a substitutivity property and that the auxiliary relation is preserved under the evaluation relation, and hence that Evaluation Bisimulation contains the auxiliary relation. In fact we prove a slightly weaker result. We show that the transitive

closure of the auxiliary relation is preserved under the evaluation relation. This is because we need symmetry which is easier to prove for the transitive closure.

Lemma 4.4

1. $E \simeq^* F \ \& \ F \simeq^o_{\Downarrow} F' \Rightarrow E \simeq^* F'$
2. $E \simeq^* E$
3. $E \simeq^o_{\Downarrow} F \Rightarrow E \simeq^* F$

Proof

1. Use transitivity of $\simeq^o_{\Downarrow}$ and the fact that we may assume that $E \simeq^* F$ is normal.
2. $E \simeq^o_{\Downarrow} E$, then use induction on structure of E.
3. Follows from 1 & 2.

□

We now prove the main result of this section; that the auxiliary relation has a substitutivity property. We already know that Evaluation Bisimulation is an equivalence relation. Then, once we have proved that Evaluation Bisimulation contains (the transitive closure of) the auxiliary relation, we will be able to read off congruence for Evaluation Bisimulation.

Lemma 4.5 *Substitutivity for* $\simeq^*$

$(E' \simeq^* F' \ \& \ E \simeq^* F) \Rightarrow E'[E/X] \simeq^* F'[F/X]$

Proof

The result follows by induction on derivation of $E' \simeq^* F'$, lemma 2.11 on page 25 and that if $E' \simeq^o_{\Downarrow} F'$ then $E'[F/X] \simeq^o_{\Downarrow} F'[F/X]$. □

We will also need a very limited form of substitutivity for $\simeq^o$.

Lemma 4.6

For any NCCS processes P_1, P_2, P_3 with $P_1 \simeq^o P_2$ and any channel n

1. $P_1P_3 \simeq^o P_2P_3$
2. $\nu n.P_1 \simeq^o \nu n.P_2$

Proof

The first part follows trivially from the definition of $\simeq_{\Downarrow}$ (definition 3.4 on page 42). The second part follows from lemma 2.14 and corollary 2.17. □

Finally we need to show that we may work our way up a derivation tree for the proof of $E \simeq^* F$ in a "structure preserving" manner. The following lemma shows this. We will use the results in section 4.3.

Lemma 4.7

For $E \simeq^* F$ simple then:

1. $\nu n.E' \simeq^* F \Rightarrow \exists E''\ (E' \simeq^* E''\ \&\ \nu n.E'' \simeq^o_{\Downarrow} F)$
2. $E_1 E_2 \simeq^* F \Rightarrow \exists E'_1, E'_2\ (E_1 \simeq^* E'_1\ \&\ E_2 \simeq^* E'_2\ \&\ E'_1 E'_2 \simeq^o_{\Downarrow} F)$
3. $l.E' \simeq^* F \Rightarrow \exists E''\ (E' \simeq^* E''\ \&\ l.E'' \simeq^o_{\Downarrow} F)$
4. $\mathbf{rec}X.E' \simeq^* F \Rightarrow \exists E''\ (E' \simeq^* E''\ \&\ \mathbf{rec}X.E'' \simeq^o_{\Downarrow} F)$
5. $E_1 + E_2 \simeq^* F \Rightarrow \exists E'_1, E'_2\ (E_1 \simeq^* E'_1\ \&\ E_2 \simeq^* E'_2\ \&\ E'_1 + E'_2 \simeq^o_{\Downarrow} F)$
6. $E_1 \simeq^* E'_1\ \&\ E_2 \simeq^* E'_2 \Rightarrow E_1 E_2 \simeq^* E'_1 E'_2$

Proof

1, 2, 3, 4 and 5 all follow directly from the last rule used in the derivation of $E \simeq^* F$.

6 follows from the definition of $\simeq^*$ (definition 4.1 on page 52) plus reflexivity of $\simeq^o_{\Downarrow}$.

□

4.3 Closure of Auxiliary Relation

In order to prove Evaluation Bisimulation contains the auxiliary relation, we need to show that the auxiliary relation is closed under evaluation, i.e. that if $E \simeq^* F$ and $E \Downarrow l.E'$ then $F \Downarrow l.F'$ with $E' \simeq^* F'$. We do not need to quantify this over all possible parallel contexts since we already know that the auxiliary relation is closed under such contexts. We proceed by first proving the result by tedious case splitting for $E \simeq^* F$ simple, and then extend the result to the general case.

Lemma 4.8

For closed NCCS processes P, P' and Q and label l then $P \Downarrow l.P'$ and $P \simeq^* Q$ simple implies that there is a Q' with $Q \Downarrow l.Q'$ and $P' \simeq^* Q'$.

Proof

We proceed by induction on the derivation of $P \Downarrow l.P'$. Use case by case analysis:

$\Downarrow$**PRE** $P = \nu\vec{n}.((l.E_1 + E_2)E_3)$

By (multiple uses of) lemmas 4.7(1) and 4.6 $\exists Q''$ s.t.

$$\nu\vec{n}.Q'' \simeq^o_{\Downarrow} Q \tag{4.1}$$

$$(l.E_1 + E_2)E_3 \simeq^* Q'' \tag{4.2}$$

Using lemma 4.7(2) and (4.2) $\exists Q_1, Q_2$ s.t.

$$Q_1Q_2 \simeq^o_{\Downarrow} Q'' \tag{4.3}$$

$$l.E_1 + E_2 \simeq^* Q_1 \tag{4.4}$$

$$E_3 \simeq^* Q_2 \tag{4.5}$$

Using lemma 4.7(5) and (4.4) $\exists Q_{11}, Q_{12}$ s.t.

$$Q_{11} + Q_{12} \simeq^o_{\Downarrow} Q_1 \tag{4.6}$$

$$l.E_1 \simeq^* Q_{11} \tag{4.7}$$

$$E_2 \simeq^* Q_{12} \tag{4.8}$$

Using lemma 4.7(3) and (4.7) $\exists Q'_{11}$ s.t.

$$l.Q'_{11} \simeq^o_{\Downarrow} Q_{11} \tag{4.9}$$

$$E_1 \simeq^* Q'_{11} \tag{4.10}$$

We must have $l.Q'_{11} \Downarrow l.Q'_{11}$ and so by (4.6), there is Q''_{11} such that

$$Q_{11} \Downarrow l.Q''_{11} \tag{4.11}$$

$$Q'_{11} \simeq_{\Downarrow} Q''_{11} \tag{4.12}$$

From (4.11) we can deduce

$$\nu\vec{n}((Q_{11} + Q_{12})Q_2) \Downarrow l.\nu\vec{n}.(Q''_{11}Q_2) \tag{4.13}$$

Then by lemma 4.6, transitivity of $\simeq^o_{\Downarrow}$ (theorem 3.7) and (4.1), (4.3) and (4.9) we derive

$$\nu\vec{n}((Q_{11}+Q_{12})Q_2) \simeq_{\Downarrow} Q \tag{4.14}$$

Hence by (4.13) and (4.14), there is some Q' such that.

$$Q \Downarrow l.Q' \tag{4.15}$$

$$\nu\vec{n}(Q''_{11}Q_2) \simeq_{\Downarrow} Q' \tag{4.16}$$

Observe that by (4.10), (4.12) and lemma 4.4(1) we have

$$E_1 \simeq^* Q''_{11} \tag{4.17}$$

So by (4.5), (4.16) and (4.17) and the definition of $\simeq^*$ (definition 4.1 on page 52) we have $\nu\vec{n}.(E_1E_3) \simeq^* Q'$. Thus $P' \simeq^* Q'$, as required.

$\Downarrow$**COMP** $P = \nu\vec{n}.(P_1P_2)$ where:

$$P_1 \Downarrow l'.P'_1 \tag{4.18}$$

$$P_2 \Downarrow \overline{l}'.P'_2 \tag{4.19}$$

$$\nu\vec{n}.(P'_1P'_2) \Downarrow l.P' \tag{4.20}$$

By (multiple use of) lemmas 4.7(1) and 4.6 $\exists Q''$ s.t.

$$\nu\vec{n}.Q'' \simeq^o_{\Downarrow} Q \tag{4.21}$$

$$P_1P_2 \simeq^* Q'' \tag{4.22}$$

Using lemma 4.7(2) on (4.22) gives:

$$Q_1Q_2 \simeq^o_{\Downarrow} Q'' \tag{4.23}$$

$$P_i \simeq^* Q_i \tag{4.24}$$

Using the induction hypothesis on (4.18), (4.19) and (4.24) gives Q'_1, Q'_2 s.t.

$$Q_1 \Downarrow l'.Q'_1 \tag{4.25}$$

$$Q_2 \Downarrow \overline{l}'.Q'_2 \tag{4.26}$$

$$P'_i \simeq^* Q'_i \tag{4.27}$$

(4.27), lemma 4.7(6) and RES give:

$$\nu\vec{n}.(P_1'P_2') \simeq^* \nu\vec{n}.(Q_1'Q_2') \tag{4.28}$$

(4.21), (4.28) and induction hypothesis show that $\exists Q'$ s.t.

$$\nu\vec{n}.(Q_1'Q_2') \Downarrow l.Q' \tag{4.29}$$

$$P' \simeq^* Q' \tag{4.30}$$

(4.25), (4.26), (4.29) and COMP gives $\nu\vec{n}.(Q_1Q_2) \Downarrow l.Q'$ and then (4.21) and (4.23) gives $Q \Downarrow l.Q'$ as required.

$\Downarrow$**RES** $P = \nu\vec{n}.(\mathrm{rec}X.E|P_2)$

Since $\nu\vec{n}.(\mathrm{rec}X.E|P_2) \simeq^* Q$, by (multiple uses of) lemmas 4.7(1) and 4.6 $\exists Q''$ s.t.

$$\nu\vec{n}.Q'' \simeq^{o}_{\Downarrow} Q \tag{4.31}$$

$$\mathrm{rec}X.E|P_2 \simeq^* Q'' \tag{4.32}$$

By lemma 4.7(2) $\exists Q_1, Q_2$ s.t.

$$Q_1Q_2 \simeq^{o}_{\Downarrow} Q'' \tag{4.33}$$

$$\mathrm{rec}X.E \simeq^* Q_1 \tag{4.34}$$

$$P_2 \simeq^* Q_2 \tag{4.35}$$

By lemma 4.7(4) and (4.34) $\exists Q_1'$ s.t.

$$E \simeq^* Q_1' \tag{4.36}$$

$$\mathrm{rec}X.Q_1' \simeq^{o}_{\Downarrow} Q_1 \tag{4.37}$$

Using REC and (4.36) gives:

$$\mathrm{rec}X.E \simeq^* \mathrm{rec}X.Q_1' \tag{4.38}$$

Then lemma 4.5, (4.36) and (4.38) gives:

$$E[\mathrm{rec}X.E/X] \simeq^* Q_1'[\mathrm{rec}X.Q_1'/X] \tag{4.39}$$

Then lemma 4.7(6), (4.35) and (4.39) gives

$$E[\mathrm{rec}X.E/X]P_2 \simeq^* Q_1'[recX.Q_1'/X]Q_2 \tag{4.40}$$

Then applying RES (multiple times) gives:

$$\nu\vec{n}.(E[\text{rec}X.E/X]P_2 \simeq^* \nu\vec{n}.(Q_1'[\text{rec}X.Q_1'/X]Q_2) \tag{4.41}$$

$P \Downarrow l.P'$ was derived from:

$$\nu\vec{n}.(E[\text{rec}X.E/X]P_2 \Downarrow l.P' \tag{4.42}$$

Induction hypothesis on (4.41) and (4.42) shows that $\exists Q'''$ s.t.

$$\nu\vec{n}.(Q_1'[\text{rec}X.Q_1'/X]Q_2) \Downarrow l.Q''' \tag{4.43}$$

$$P' \simeq^* Q''' \tag{4.44}$$

(4.43) and $\Downarrow$REC implies:

$$\nu\vec{n}.(\text{rec}X.Q_1'|Q_2) \Downarrow l.Q''' \tag{4.45}$$

(4.31), (4.33) and (4.37) and lemma 4.6 gives:

$$\nu\vec{n}.(\text{rec}X.Q_1'|Q_2) \simeq^o_{\Downarrow} Q \tag{4.46}$$

(4.45) and (4.46) implies $Q \Downarrow l.Q'$ with:

$$Q''' \simeq^o_{\Downarrow} Q' \tag{4.47}$$

Then (4.41), (4.47) and lemma 4.5(1) implies $P' \simeq^* Q'$ as required.

$\Downarrow$STRUC

$\exists P_1, P_2$ s.t. $P_1 \equiv P$, $P' \equiv P_2$ and

$$P_1 \Downarrow l.P_2 \tag{4.48}$$

Now $P \equiv P_1 \Rightarrow P_1 \simeq^* Q$ hence 4.48 and induction hypothesis give Q' s.t.

$$P_2 \simeq^* Q' \tag{4.49}$$

$$Q \Downarrow l.Q' \tag{4.50}$$

$P' \equiv P_2 \Rightarrow P' \simeq^* P_2 \overset{(4.49)}{\Longrightarrow} P' \simeq^* Q'$ as required.

□

Corollary 4.9

For closed NCCS processes P, P' and Q and label l then $P \Downarrow l.P'$ and $P \simeq^* Q$ implies that there is a Q' with $Q \Downarrow l.Q'$ and $P' \simeq^* Q'$.

Proof

If $P \simeq^* Q$ is simple then the result follows from lemma 4.8. Otherwise the last rule used was (STRUC) (since we may assume by lemma 4.3 that it is normal). Hence there exists $P'' \equiv P$ and $P'' \simeq^* Q$ and this is simple. Also $P'' \Downarrow l.P'$ using ($\Downarrow$STRUC). Hence using lemma 4.8 we get $Q \Downarrow l.Q'$ and $P' \simeq^* Q'$ as required. □

4.4 Symmetry of the Auxiliary Relation

Due to the fact that Evaluation Bisimulation is a *bisimulation* and not just a *simulation*, we need to be able to prove that, for $E \simeq^* F$, the auxiliary relation is closed under reductions of both E and F. Showing that $\simeq^*$ is closed under reductions for E has been shown in corollary 4.9. Showing that $\simeq^*$ is closed under reductions for F is extremely hard. Instead we follow Howe in [How96] and prove that the auxiliary relation is symmetric under transitive closure. This is straightforward, proceeding by a simple case split.

Definition 4.10

For any binary relation $\mathcal{R}$, the transitive closure $\mathcal{R}^{tc}$ is the binary relationship generated by $\mathcal{R}$ under the following properties:

$$\frac{}{P\mathcal{R}^{tc}P} \qquad \frac{P\mathcal{R}^{tc}P' \quad P'\mathcal{R}P''}{P\mathcal{R}^{tc}P''}$$

Lemma 4.11

$E \simeq^* F \Rightarrow F{\simeq^*}^{tc}E$

Proof

We proceed by induction of the derivation of $E \simeq^* F$ using the symmetry of $\simeq^o_{\Downarrow}$ (theorem 3.7 on page 43) . We use lemma 4.4 on page 54 many times without comment. Last rule used was:

NULL By lemma 4.4(3) and symmetry of $\simeq^o_{\Downarrow}$.

VAR Similar to previous case.

COM $E = \alpha.G$ implies that there is an E' s.t.

$$G \simeq^* E' \tag{4.51}$$

$$l.E' \simeq^o_{\Downarrow} F \tag{4.52}$$

Then by induction hypothesis on 4.51 implies that

$$E' \simeq^{*tc} G \tag{4.53}$$

and then using (COM) multiple times implies that

$$l.E' \simeq^{*tc} l.G \tag{4.54}$$

Then symmetry of $\simeq^o_{\Downarrow}$ and 4.52 imply that

$$F \simeq^o_{\Downarrow} l.E' \tag{4.55}$$

and then by lemma 4.4(3) we derive that

$$F \simeq^* l.E' \tag{4.56}$$

Then 4.56 and 4.54 imply that $F \simeq^{*tc} \alpha.G = E$, as required.

REC $E = \mathrm{rec}X.G$ implies that there is an E' s.t.

$$G \simeq^* E' \tag{4.57}$$

$$\mathrm{rec}X.E' \simeq^o_{\Downarrow} F \tag{4.58}$$

Then by induction hypothesis on 4.57 and then (REC) multiple times we get

$$\mathrm{rec}X.E' \simeq^{*tc} \mathrm{rec}X.G \tag{4.59}$$

By symmetry of $\simeq^o_{\Downarrow}$ on 4.58 and then lemma 4.4(3) we may derive that

$$F \simeq^* \mathrm{rec}X.E' \tag{4.60}$$

But 4.60 and 4.59 imply that $F \simeq^{*tc} \mathrm{rec}X.G = E$, as required.

RES $E = \nu n.G$ implies that there is an E' s.t.

$$G \simeq^* E' \tag{4.61}$$

$$\nu n.E' \simeq^o_{\Downarrow} F \tag{4.62}$$

Then by induction hypothesis on 4.61 and then (RES) multiple times we get

$$\nu n.E' \simeq^{*tc} \nu n.G \tag{4.63}$$

By symmetry of $\simeq^{o}_{\Downarrow}$ on 4.62 and then lemma 4.4(3) we may derive that

$$F \simeq^{*} \nu n.E' \tag{4.64}$$

But 4.64 and 4.63 imply that $F \simeq^{*tc} \nu n.G = E$, as required.

SUM $E = E_1 + E_2$ implies that there are E'_1, E'_2 s.t.

$$E'_1 + E'_2 \simeq^{o}_{\Downarrow} F \tag{4.65}$$

$$E_1 \simeq^{*} E'_1 \tag{4.66}$$

$$E_2 \simeq^{*} E'_2 \tag{4.67}$$

By symmetry of $\simeq^{o}_{\Downarrow}$ on 4.65 and then lemma 4.4(3) we may derive that

$$F \simeq^{*} E'_1 + E'_2 \tag{4.68}$$

The induction hypothesis on 4.66 and 4.67 imply that

$$E'_1 \simeq^{*tc} E_1 \tag{4.69}$$

$$E'_2 \simeq^{*tc} E_2 \tag{4.70}$$

Then (SUM) multiple times on 4.69 and 4.70 implies that

$$E'_1 + E'_2 \simeq^{*tc} E_1 + E_2 \tag{4.71}$$

Then 4.71 and 4.68 imply that $F \simeq^{*tc} E_1 + E_2 = E$, as required.

COMP $E = E_1 E_2$ implies that there are E'_1, E'_2 s.t.

$$E'_1 E'_2 \simeq^{o}_{\Downarrow} F \tag{4.72}$$

$$E_1 \simeq^{*} E'_1 \tag{4.73}$$

$$E_2 \simeq^{*} E'_2 \tag{4.74}$$

By symmetry of $\simeq^{o}_{\Downarrow}$ on 4.72 and then lemma 4.4(3) we may derive that

$$F \simeq^{*} E'_1 E'_2 \tag{4.75}$$

The induction hypothesis on 4.73 and 4.74 imply that

$$E_1' \simeq^{*tc} E_1 \tag{4.76}$$

$$E_2' \simeq^{*tc} E_2 \tag{4.77}$$

Then using (COMP) multiple times on 4.76 and 4.77 implies that

$$E_1' E_2' \simeq^{*tc} E_1 E_2 \tag{4.78}$$

Then 4.78 and 4.75 imply that $F \simeq^{*tc} E_1 E_2 = E$, as required.

STRUC $E \equiv E'$ implies that

$$E' \simeq^{*} F \tag{4.79}$$

$$E \equiv E' \tag{4.80}$$

Then by induction hypothesis on 4.79 we derive that

$$F \simeq^{*tc} E' \tag{4.81}$$

Symmetry of $\equiv$, then lemma 4.4(2) and then (STRUC) gives

$$E' \simeq^{*} E \tag{4.82}$$

Then 4.82 and 4.81 imply that $F \simeq^{*tc} E$, as required.

□

Corollary 4.12

1. $P \Downarrow l.P' \;\; \& \;\; P \simeq^{*tc} Q \Rightarrow \exists Q' \; (Q \Downarrow l.Q' \;\; \& \;\; P' \simeq^{*tc} Q')$
2. $Q \Downarrow l.Q' \;\; \& \;\; P \simeq^{*tc} Q \Rightarrow \exists P' \; (P \Downarrow l.P' \;\; \& \;\; P' \simeq^{*tc} Q')$

Proof

1. Follows from Corollary 4.9 and the definition of transitive closure.
2. Follows from (1) and Lemma 4.11.

□

4.5 Congruence for Evaluation Bisimulation

Having derived the difficult results we may easily read off the equivalence of Evaluation Bisimulation and the auxiliary relation, and hence the desired congruence property.

Corollary 4.13

$$\simeq^{*tc} \subseteq \{\simeq^{*tc}\}$$

Proof

This follows directly from the definition of Evaluation Bisimulation (definition 3.4 on page 42), lemma 4.5 and Corollary 4.12 overleaf.

□

Restatement of Theorem 3.7

1. $\simeq^o_{\Downarrow}$ is an equivalence relation.
2. $\simeq^o_{\Downarrow}$ is a congruence relation.
3. $\simeq_{\Downarrow}$ coincides with $\simeq'_{\Downarrow}$.

Proof of Theorem 3.7(2)

Lemma 4.4 implies that $\simeq^o_{\Downarrow} \subseteq \simeq^*$. Then by the definition of transitive closure we see that $\simeq^* \subseteq \simeq^{*tc}$. For any closed NCCS processes P and Q with $P \simeq^{*tc} Q$ then by corollary 4.13 we see that $P \simeq_{\Downarrow} Q$. But for any (not necessarily closed) NCCS processes P and Q with free variables X_i and $P \simeq^{*tc} Q$ then by lemma 4.5 for any closed NCCS processes R_i we may deduce that $P[R_i/X_i] \simeq^{*tc} Q[R_i/X_i]$. Hence we see that $P \simeq^o_{\Downarrow} Q$ and hence $\simeq^* \subseteq \simeq^{*tc} \subseteq \simeq^o_{\Downarrow}$. Therefore $\simeq^o_{\Downarrow} \equiv \simeq^*$.

We already know that $\simeq^o_{\Downarrow}$ is an equivalence relation, and now we have that it is closed under the various NCCS constructs (because $\simeq^*$ is), hence $\simeq^o_{\Downarrow}$ is a congruence relation. □

Chapter 5

Link between evaluation and transition

As has been noted before, it has been traditional to define reductions on calculi, particularly those used for concurrency, in terms of a (labelled) transition system. To show how NCCS relates to other calculi we now present the relationship between evaluation on NCCS and transition on τNCCS. This will also help our understanding of what evaluation is and show that it is just a big step version of transition.

We firstly look at the translation of the transition relation on τNCCS into the evaluation relation on NCCS, seeing that any delay-like[1] sequence of transitions may be translated into a single evaluation. We then examine the translation of the evaluation relation on NCCS into the transition relation on τNCCS and conclude, using the properties of strong CCS equivalence, that all evaluations may be seen as a delay-like sequence of transitions modulo strong equivalence.

5.1 Transition to Evaluation

Recall from section 2.6 on page 35 the definitions of the translations, $(-)^t$ and $(-)^T$, between NCCS and τNCCS.

[1] A sequence of transitions is delay-like if it consists of zero or more silent actions followed by an input or output.

We would like to show that if $P \xrightarrow{\tau^* l} P'$ then $P^t \Downarrow l.P'^t$. However this is not quite true. In fact $P^t \Downarrow l.R$ for $R = P'^t$ or $R = (\nu n.\overline{n}.0)P'^t$. In order to show this we first prove that $P \xrightarrow{l} P'$ implies that $P^t \Downarrow l.R$, with R as above. We then extend the result. Both steps follow by simple case splitting.

Lemma 5.1

If $P \xrightarrow{l} P'$ in τNCCS then $P^t \Downarrow l.R$ in NCCS, where $R = P'^t$ or $R = (\nu n.\overline{n}.0)P'^t$.

Proof

We proceed by induction on the derivation of $P \xrightarrow{l} P'$.

$\rightarrow$**ACT** $P = l.P'$

Then $P^t = l.(P')^t \Downarrow l.(P')^t$ by ($\Downarrow$PRE)

$\rightarrow$**COM$_1$** $P = P_1P_2$, $P' = P_1'P_2$ and $P_1 \xrightarrow{l} P_1'$. By induction hypothesis:

$$(P_1)^t \Downarrow l.R_1$$

so by lemma 2.14, for any fresh name n:

$$(\nu n.(P_1P_2))^t = \nu n.((P_1)^t(P_2)^t) \Downarrow l.(\nu n.(R_1(P_2)^t))$$

Then using ($\Downarrow$STRUC) we get

$$(P_1P_2)^t = (P_1)^t(P_2)^t \Downarrow l.(R_1(P_2)^t)$$

Then if $R = P_1'^t$ we have

$$R(P_2)^t \equiv (P_1'P_2)^t = (P')^t$$

otherwise

$$R(P_2)^t \equiv (\nu n.\overline{n}.0)(P_1'P_2)^t = (\nu n.\overline{n}.0)(P')^t$$

so $(P_1P_2)^t \Downarrow l.R$ as required.

$\rightarrow$**COM$_2$** Follows symmetrically.

$\rightarrow$**COM$_3$** Cannot occur since $l \neq \tau$.

$\rightarrow$**SUM** $P = \Sigma\alpha_i.P_i, l = \alpha_j, P' = P_j$ and $l.P' \xrightarrow{l} P'$.

We case split again according to whether any of the α_i are τ or not. If not then $(P)^t \equiv l_j.(P_j)^t + Q$, where Q is the (translation of the) rest of the sum. Then by ($\Downarrow$PRE) and ($\Downarrow$STRUC) we have $(P)^t \Downarrow l_j.(P_j)^t$ as required.

Otherwise $(P)^t \equiv \nu n.(\overline{n}.0|(l_j.P_j + Q))$ where Q is the (translation of the) rest of the sum and n is a fresh name. Then $(P)^t \Downarrow l.\nu n.(\overline{n}.0|P')$ by ($\Downarrow$STRUC) and ($\Downarrow$PRE). But $\nu n.(\overline{n}.0|P') \equiv (\nu m.\overline{m}.0)(P')^t$ and so $(P)^t \Downarrow (\nu m.\overline{m}.0)(P')^t$ as required.

$\rightarrow$**RES** $P = \nu n.P_1, P' = \nu n.P_1'$ and $P_1 \xrightarrow{l} P_1' \quad (l, \overline{l} \neq n)$

By induction hypothesis $(P_1)^t \Downarrow l.R_1$, so by lemma 2.14 $(P)^t \Downarrow l.\nu n.R_1$. But $\nu n.R_1 \equiv (\nu m.\overline{m}.0)\nu n.P_1$ and using ($\Downarrow$STRUC) $P^t \Downarrow l.R$ as required.

$\rightarrow$**REC** $P = \mathrm{rec}X.E$ and $E[P/X] \xrightarrow{l} P'$

By induction hypothesis:

$$(E[P/X])^t \Downarrow l.R$$

But by lemma 2.26:

$$(E[P/X])^t \equiv (E)^t[(P)^t/X] = (E)^t[\mathrm{rec}X.(E)^t/X]$$

So $(P)^t = \mathrm{rec}X.(E)^t \Downarrow l.R$ using ($\Downarrow$REC).

□

Lemma 5.2

$P \xrightarrow{\tau} P'$ in τNCCS and $\nu\vec{n}.((P')^tQ) \Downarrow l.Q'$ in NCCS, then $\nu\vec{n}.((P)^tQ) \Downarrow l.Q'$.

Proof

We proceed by induction on the derivation of $P \xrightarrow{\tau} P'$.

$\rightarrow$**ACT** $P = \tau.P'$

So $(P)^t = \nu m.(\overline{m}.0|m.(P')^t)$ where $m \notin \mathrm{fn}(P', Q, Q') \cup \{\vec{n}\}$. Then we have the following derivation:

$$\dfrac{\overline{\overline{m}.0 \Downarrow \overline{m}.0} \qquad \overline{m.(P')^t|Q \Downarrow m.((P')^tQ)} \qquad \dfrac{\vdots\ I.H.}{\nu\vec{n}.((P')^tQ) \Downarrow l.Q'}}{\nu\vec{n}m.(\overline{m}.0|m.(P')^t|Q) \Downarrow l.\nu m.Q'}$$

But

$$\nu\vec{n}m.(\overline{m}.0|m.(P')^t|Q) \equiv \nu\vec{n}.((P)^tQ)$$

The result then follows by noting that $\nu m.Q' \equiv Q'$ since $m \notin \mathrm{fn}(Q')$.

$\rightarrow$**COM**$_1$ $P = P_1P_2$, $P' = P_1'P_2$ and $P_1 \xrightarrow{\tau} P_1'$

If $\nu\vec{n}.((P')^tQ) \Downarrow l.Q'$ then $\nu\vec{n}.((P_1')^t((P_2)^tQ)) \Downarrow l.Q'$ so by induction hypothesis on $P \xrightarrow{\tau} P_1'$

$$\nu\vec{n}.((P_1)^t((P_2)^tQ)) \Downarrow l.Q'$$

i.e. $\nu\vec{n}.((P)^tQ) \Downarrow l.Q'$.

$\rightarrow$**COM**$_2$ symmetrically.

$\rightarrow$**COM**$_3$ $P = P_1P_2$, $P' = P_1'P_2'$, $P_1 \xrightarrow{l'} P_1'$ and $P_2 \xrightarrow{\overline{l'}} P_2'$.

By lemma 5.1 $(P_1)^t \Downarrow l'.(P_1')^t$ and $(P_2)^t \Downarrow \overline{l'}.(P_2')^t$. So if

$$\nu\vec{n}.((P_1')^t(P_2')^tQ) \Downarrow l.Q'$$

then by $\Downarrow$COMP and Lemma 2.14 we get the required result:

$$\nu\vec{n}.((P_1)^t(P_2)^tQ) \Downarrow l.Q'$$

$\rightarrow$**SUM** $P = \sum \alpha_i.P_i$, $P' = P_j$, $\alpha_j = \tau$ and $P \xrightarrow{\tau} P'$.

Then for some fresh name m (hence also different from all $\vec{n}$)

$$(P)^t \equiv \nu m.(\overline{m}.0(m.(P')^t + R))$$

where R is the translation of the rest of the sum. Then the result follows from the following derivation and by noting that $m \notin \mathrm{fn}(Q')$ and $\nu\vec{n}m.(\overline{m}.0|(m.(P')^t + R)Q) \equiv \nu\vec{n}.((P)^tQ)$.

$$\dfrac{\dfrac{}{\overline{m}.0 \Downarrow \overline{m}.0} \quad \dfrac{}{(m.(P')^t + R)|Q \Downarrow m.((P')^tQ)} \quad \dfrac{\begin{array}{c}\vdots\ I.H. \\ \nu\vec{n}.((P')^tQ) \Downarrow l.Q'\end{array}}{\nu\vec{n}m.(0(P')^tQ) \Downarrow l.\nu m.Q'}}{\nu\vec{n}m.(\overline{m}.0(m.(P')^t + R)Q) \Downarrow l.\nu m.Q'}$$

→RES $P = \nu m.P_1$, $P' = \nu m.P_1'$ and $P_1 \xrightarrow{\tau} P_1'$

If $\nu\vec{n}.((P')^t Q) \Downarrow l.Q'$ and $\nu\vec{n}.((P')^t Q) = \nu\vec{n}m.((P_1')^t Q)$ then by induction hypothesis on $P_1 \xrightarrow{\tau} P_1'$,

$$\nu\vec{n}m.((P_1)^t Q) \Downarrow l.Q'$$

Therefore $\nu\vec{n}.((P)^t Q) \Downarrow l.Q'$.

→REC $P = \mathrm{rec}X.E$ and $E[P/X] \xrightarrow{\tau} P'$

By induction hypothesis, if $\nu\vec{n}.((P')^t Q) \Downarrow l.Q'$ then we get

$$\nu\vec{n}.((E[P/X])^t Q) \Downarrow l.Q'$$

Hence

$$\nu\vec{n}.((E)^t[\mathrm{rec}X.(E)^t/X]Q) \Downarrow l.Q'$$

Therefore we get

$$\nu\vec{n}.((\mathrm{rec}X.(E)^t)Q) \Downarrow l.Q'$$

i.e. $\nu\vec{n}.((P)^t Q) \Downarrow l.Q'$.

□

Theorem 5.3

If $P \xrightarrow{\tau^* l} P'$ in τNCCS, then in NCCS $P^t \Downarrow l.R$ where $R = (P')^t$ or $R = (\nu m.\overline{m}.0)(P')^t$.

Proof

Combine lemmas 5.1 and 5.2. □

We may observe that $\nu m.\overline{m}.0$ is essentially the same as the null process 0. Then we may see that the evaluation relation is a "delay" reduction relation, in that the evaluation relation ignores silent actions before a visible action, but not afterwards.

5.2 Evaluation to Transition

Recall from [Mil89] that CCS strong equivalence, $\sim_\rightarrow$ (definition 3.15 on page 47), is a congruence relation (section 4.4 on pages 97–101 of [Mil89]).

We will use this frequently and without comment in the proof of the next theorem.

Theorem 5.4

If $P \Downarrow l.P'$ in NCCS then $P \xrightarrow{\tau^* l} Q$ in τNCCS for some $Q \sim_{\rightarrow} P'$. By lemma 2.24(2) we may assume that $Q \in$ NCCS.

Proof

We proceed by induction on the derivation of $P \Downarrow \alpha.P'$

$\Downarrow$**PRE** $P = \nu\vec{n}.((l.P_1 + P_3)P_2)$ and $P' = \nu\vec{n}.P_1|P_2$ where $l, \bar{l} \notin \{\vec{n}\}$.

Then we get

$$P = \nu\vec{n}.((l.P_1 + P_3)P_2) \xrightarrow{l} \nu\vec{n}.(P_1P_2)$$

by $(\rightarrow ACT)$, $(\rightarrow SUM_1)$, $(\rightarrow COMM_1)$ and $(\rightarrow RES)^*$. Therefore we take $Q = \nu\vec{n}.(P_1P_2) = P'$.

$\Downarrow$**COMP** $P = \nu\vec{n}.(P_1P_2)$, $P_1 \Downarrow l'.P_1'$, $P_2 \Downarrow \bar{l}'.P_2'$ and $\nu\vec{n}.(P_1'P_2') \Downarrow l.P'$.

By induction hypothesis:

$$\begin{array}{rcccl} P_1 & \xrightarrow{\tau^* l'} & Q_1 & \sim_{\rightarrow} & P_1' \\ P_2 & \xrightarrow{\tau^* \bar{l}'} & Q_2 & \sim_{\rightarrow} & P_2' \\ \nu\vec{n}.(P_1'P_2') & \xrightarrow{\tau^* l} & Q' & \sim_{\rightarrow} & P' \end{array}$$

For some Q_1, Q_2 and Q'. Then:

$$P_1P_2 \xrightarrow{\tau^+} Q_1Q_2 \sim_{\rightarrow} P_1'P_2'$$

so

$$\nu\vec{n}.(P_1P_2) \xrightarrow{\tau^+} \nu\vec{n}.(Q_1Q_2) \sim_{\rightarrow} \nu\vec{n}.(P_1'P_2')$$

hence

$$\nu\vec{n}.(P_1P_2) \xrightarrow{\tau^+ l} Q$$

for some $Q \sim_{\rightarrow} Q' \sim_{\rightarrow} P'$.

$\Downarrow$**REC** $P = \nu\vec{n}.((\mathrm{rec}X.E)P_2)$ and $\nu\vec{n}.(E[\mathrm{rec}X.E/X]P_2) \Downarrow l.P'$

Hence by induction hypothesis

$$\nu\vec{n}.(E[\mathrm{rec}X.E/X]P_2) \xrightarrow{\tau^* l} Q' \sim_{\rightarrow} P'$$

for some Q'. But

$$\nu\vec{n}.(E[\mathrm{rec}X.E/X]P_2) \sim_\rightarrow \nu\vec{n}.((\mathrm{rec}X.E)P_2) = P$$

Hence we get $P \xrightarrow{\tau^* l} Q \sim_\rightarrow Q' \sim_\rightarrow P'$ for some Q.

$\Downarrow$**STRUC** $P \equiv P_1$, $P_1 \Downarrow l.P_1'$ and $P_1' \equiv P'$.

By induction hypothesis:

$$P_1 \xrightarrow{\tau^* l} Q_1 \sim_\rightarrow P_1'$$

for some Q_1 and by lemma 2.24(3) $P_1' \sim_\rightarrow P'$ and $P \sim_\rightarrow P_1$. So we get:

$$P \xrightarrow{\tau^* l} Q \sim_\rightarrow Q_1 \sim_\rightarrow P_1' \sim_\rightarrow P'$$

□

5.3 Summary of relationship

We now summarise the results of this chapter because they will be used repeatedly in the next chapter. We also give various conclusions from the above theorems that will be useful in different cases.

Theorem 5.5

1. For P, Q NCCS processes with $P \Downarrow l.Q$, there exists $Q' \in \tau$NCCS with $Q \sim_\rightarrow Q'$ and $P \xrightarrow{\tau^* l} Q'$.
2. For P, Q τNCCS processes with $P \xrightarrow{\tau^* l} Q$ then
   (a) there exists Q' where Q' is either Q^t or $(\nu n.\overline{n}.0)Q^t$ and $P^t \Downarrow l.Q'$.
   (b) there exists Q' with $Q' \simeq_\Downarrow Q^t$, $Q' \sim_\rightarrow Q$ and $P^t \Downarrow l.Q'$.

Proof

Part 1 is just theorem 5.4. Then (a) is just theorem 5.3. (b) follows in two parts. Firstly we note that $(\nu n.\overline{n}.0)P \sim_\rightarrow P$ for all P (and remember from note 2.21 that we do not bother writing the translation from NCCS to τNCCS). The other part will follow from the fact that $P \equiv Q$ implies $P \simeq_\Downarrow Q$[2] and from lemma 6.1 in the next chapter. □

[2]This is easy to see by taking $\mathcal{R} = \{(P,Q) \text{ s.t. } P \equiv Q\}$ and noting that if $P \Downarrow l.P'$ then $Q \Downarrow l.P'$ by ($\Downarrow$STRUC) and that $P \equiv P$. Hence $\mathcal{R}$ is a post-fixed point of definition 3.4 on page 42.

Chapter 6

Relationships between the bisimilarities

In chapter 3 various bisimulations were presented. We now look at the relationships between them and using these relationships we derive the proofs that the bisimulations are all congruences.

6.1 Inclusions between various relations

We first consider the relationship between strong CCS equivalence and evaluation bisimulation. From chapter 5 we may well expect that strong CCS equivalence (defined in definition 3.15 on page 47) is contained in evaluation bisimulation (defined in definition 3.4 on page 42) and this is indeed correct.

Lemma 6.1

1. $P \sim_{\rightarrow} Q \Rightarrow (P)^t \simeq_{\Downarrow} (Q)^t$
2. $P \sim_{\rightarrow} Q \Rightarrow (P)^t \simeq'_{\Downarrow} (Q)^t$

Proof

We only use those parts of theorem 5.5 on page 71 that we have already proved and not the part that depends on this lemma.

Let $\mathcal{R} = \{(P, Q) \in \text{NCCS} \mid P \sim_{\rightarrow} Q\}$. Then if $P\mathcal{R}Q$ and $PR \Downarrow l.P'$ then by theorem 5.5(1) $PR \xrightarrow{\tau^* l} P'' \sim_{\rightarrow} P'$, for some $P'' \in \text{NCCS}$. Then by theorem

3.17 and $P \sim_\rightarrow Q$ we have $QR \xrightarrow{\tau^* l} Q''$ for some $Q'' \sim_\rightarrow P''$ and so by theorem 5.5(2a) we get $QR = (QR)^t \Downarrow l.Q'$ with $Q' \sim_\rightarrow (Q'')^t$. Then by lemma 2.24(2) we get $Q' \sim_\rightarrow (Q'')^t \sim_\rightarrow Q'' \sim_\rightarrow P'' \sim_\rightarrow P'$ and hence that $(P', Q') \in \mathcal{R}$.

The symmetric case is identical. Thus $\mathcal{R} \subseteq \{\mathcal{R}\}$ and so $\sim_\rightarrow \subseteq \simeq_\Downarrow$. Hence the result follows. The second case follows similarly. □

We continue this rather short chain by showing that evaluation bisimulation is contained in weak bisimulation.

6.2 Coincidence of weak bisimulations

We first prove that $\approx_\Downarrow$ is identical to $\approx_\rightarrow$. Recall that theorem 3.21 on page 48 states that $\approx_\rightarrow$ is a congruence relation that is identical to $\approx_\Downarrow$.

Restatement of Theorem 3.21

1. $\approx_\rightarrow$ is a congruence relation,
2. For NCCS processes P and Q and τNCCS processes R and S then $P \approx_\Downarrow Q$ implies that $P^T \approx_\rightarrow Q^T$ and $R \approx_\rightarrow S$ implies that $R^t \approx_\Downarrow S^t$. Thus $\approx_\Downarrow$ and $\approx_\rightarrow$ coincide under translation.

Proof of Theorem 3.21(2)

$P \approx_\Downarrow Q \Rightarrow P \approx_\rightarrow Q$

Let $\mathcal{S} = \{(P, Q) \in \tau\text{NCCS} \times \tau\text{NCCS} \mid P^t \approx_\Downarrow Q^t\}$.

If $P \xrightarrow{l} P'$ and $(P, Q) \in \mathcal{S}$ then

$$P|(\bar{l}.n.0) \xrightarrow{\tau n} P' \quad n \notin \text{fn}(P, P') \tag{6.1}$$

Then by theorem 5.5(2b) there is a P'' with $P^t(\bar{l}.n.0) \Downarrow n.(P')^t$ and $P'' \simeq_\Downarrow (P')^t$. However we have $P^t \approx_\Downarrow Q^t$ so we get

$$Q^t|(\bar{l}.n.0) \Downarrow n.Q'' \tag{6.2}$$

with $(P')^t \simeq_\Downarrow P'' \approx_\Downarrow Q''$. Therefore by theorem 5.5(1) $Q^t|\bar{l}.n.0 \xrightarrow{\tau^* n} Q'''$ with $Q''' \sim_\rightarrow Q''$. But $Q^t \sim_\rightarrow Q$ (by lemma 2.24(2)) so there is a Q' with

$$Q|\bar{l}.n.0 \xrightarrow{\tau^* n} Q'|0 \tag{6.3}$$

and $Q''' \sim_\to Q'$ (because $\to$ is structure preserving and n is not free in Q). But $Q'|0 \sim_\to Q'$ and so $Q' \sim_\to Q''$. Then by lemma 6.1 and theorem 3.11 we have $Q'^t \approx_\Downarrow Q''^t$. Transitivity and symmetry of $\approx_\Downarrow$ gives $P'^t \approx_\Downarrow Q'^t$, so $(P', Q') \in \mathcal{S}$. Then (6.3) shows that

$$Q|\bar{l}.0 \stackrel{\tau^*}{\to} Q'|0 \tag{6.4}$$

because $\to$ is structure preserving and n is not a free name of P and hence not a free name of Q. Similarly we may see from (6.4) that $Q \stackrel{\tau^* l \tau^*}{\longrightarrow} Q'$ as required.

If $P \stackrel{\tau}{\to} P'$ and $(P, Q) \in \mathcal{S}$ then

$$P|(n.0) \stackrel{\tau n}{\longrightarrow} P' \quad n \notin \mathrm{fn}(P, P') \tag{6.5}$$

Then by theorem 5.5(2b) there exists P'' with $P^t|(n.0) \Downarrow n.P''$ and $(P')^t \simeq_\Downarrow P''$. Since we have $P^t \approx_\Downarrow Q^t$ there is a Q'' with $Q^t|(n.0) \Downarrow n.Q''$ and $P'' \approx_\Downarrow Q''$. Hence by theorem 5.5(1) there is a Q' with $Q|n.0 \stackrel{\tau^* n}{\longrightarrow} Q'|0$ and $Q' \sim_\to Q''$. Thus $Q \stackrel{\tau^*}{\longrightarrow} Q'$. But $Q' \sim_\to Q''$ implies (by lemma 6.1 and theorem 3.11) that $Q'^t \approx_\Downarrow Q''^t$. Then using transitivity and symmetry of $\approx_\Downarrow$ we get $P'^t \approx_\Downarrow Q'^t$ as required.

The symmetric cases follow in the same way because both relations are symmetric. Hence $[\mathcal{S}]_w \subseteq \mathcal{S}$ and so $P \approx_\Downarrow Q \Rightarrow P \approx_\to Q$ as required.

$P \approx_\to Q \Rightarrow P \approx_\Downarrow Q$

Let $\mathcal{S} = \{(P, Q) \in \mathrm{NCCS} \times \mathrm{NCCS} \mid P \approx_\to Q\}$.

If $P|(n.0) \Downarrow n.P'$, $n \notin \mathrm{fn}(P, P')$ and $(P, Q) \in \mathcal{S}$ then by theorem 5.5(1) $P(n.0) \stackrel{\tau^* n}{\longrightarrow} P''|0$ with $P' \sim_\to P''$. Therefore $P \stackrel{\tau^*}{\longrightarrow} P''$ and so $Q \stackrel{\tau^*}{\longrightarrow} Q''$ with $P'' \approx_\to Q''$. Hence $Q|(n.0) \stackrel{\tau^* n}{\longrightarrow} Q''|0$ and so by theorem 5.5(2b) there is a Q' with $Q|(n.0) \Downarrow n.Q'$ and $Q' \sim_\to Q''$. But $Q' \sim_\to Q''$ implies that $Q'' \approx_\to Q'$ and so using transitivity of $\approx_\to$ (twice) we get $P' \approx_\to Q'$ as required.

If $P|(\bar{l}.n.0) \Downarrow n.P'$ with $n \notin \mathrm{fn}(P, P')$ and $(P, Q) \in \mathcal{S}$ then by theorem 5.5(1) $P|(\bar{l}.n.0) \stackrel{\tau^* n}{\longrightarrow} P''|0$ with $P' \sim_\to P''$. Hence $P \stackrel{\tau^* l \tau^*}{\longrightarrow} P''$ and so $Q \stackrel{\tau^* l \tau^*}{\longrightarrow} Q''$ with $P'' \approx_\to Q''$. Hence $Q|(\bar{l}.n.0) \stackrel{\tau^* n}{\longrightarrow} Q''|0$ and so by theorem

5.5(2b) there exists a Q' with $Q'' \sim_{\rightarrow} Q'$ and $Q|(\bar{l}.n.0) \Downarrow n.Q'$. As above this then implies that $P' \approx Q'$ as required.

The symmetric cases follow in the same manner, since both relations are symmetric. Hence $\{\mathcal{S}\}_w \subseteq \mathcal{S}$ and so $P \approx_{\rightarrow} Q \Rightarrow P \approx_{\Downarrow} Q$ as required.

Hence $\forall P, Q \in \text{NCCS}\ \ P \approx_{\Downarrow} Q \Leftrightarrow P \approx_{\rightarrow} Q$. □

Restatement of Theorem 3.10

$\approx_{\Downarrow}$ is a congruence relation.

Proof of theorem 3.10

Theorem 3.21 tells us that $\approx_{\rightarrow}$ is a congruence and that $\approx_{\Downarrow}$ and $\approx_{\rightarrow}$ coincide. Then theorem 2.27 gives the result.

□

6.3 Equivalence of weak and barbed bisimulation

We now turn to examining barbed bisimulation. The barbed bisimulation we have defined is weak in style since we have a "big step" evaluation semantics. Hence we should not expect it to be equivalent to strong CCS equivalence, as is the case when we express it in terms of a "small step" transition relation in which we may count the number of τ-steps. However we can prove that it is the same as CCS observation equivalence. We do this by first showing that it is equivalent to weak $\Downarrow$-bisimulation. We first show that it contains weak bisimulation and then the converse. In doing this we end up showing that we do not need to use all possible contexts and can restrict them as described in theorem 3.14 on page 46.

We first prove a useful lemma.

Lemma 6.2

For any NCCS processes P, P' and fresh name n then $P|\bar{l}.n.0 \Downarrow n.P'$ implies that there exists a P'' with $P \Downarrow l.P''$.

Proof

If $P|\bar{l}.n.0 \Downarrow n.P'$ then by theorem 5.5(1) there is a Q with $Q \sim_{\rightarrow} P'$ and $P|\bar{l}.n.0 \xrightarrow{\tau^i n} Q|0$ for some i (since $\rightarrow$ is structure preserving and n is not free

in P). If we let $P_0 = P|\bar{l}.n.0$, $P_{i+1} = Q|0$ and the other P_i be the intermediate steps then there must be a j less than i s.t. P_j cannot input on channel n but P_{j+1} can. Hence $P_j = P'_j|\bar{l}.n.0$ and $P_{j+1} = P'_{j+1}|n.0$. Therefore $P \xrightarrow{\tau^*} P'_j$ and $P'_j \xrightarrow{l} P'_{j+1}$. Therefore by theorem 5.5(2b) and ($\Downarrow$STRUC) there is a P'' with $P \Downarrow l.P''$ as required. □

Lemma 6.3

$$P \approx_{\Downarrow} Q \Rightarrow P \approx^b_{\Downarrow} Q$$

Proof

Let $\mathcal{R} = \{(P,Q) \in \text{NCCS} \times \text{NCCS} \mid P \approx_{\Downarrow} Q\}$. Then for each $(P,Q) \in \mathcal{R}$ we deal with the different cases:

For all $C[-]$, if $C[P] \Downarrow$ then there exists l, P' s.t. $C[P] \Downarrow l.P'$. Hence $C[P](\bar{l}.n.0) \Downarrow n.P'$ with $n \notin \text{fn}(P, P', Q, C)$. Then $C[P] \approx_{\Downarrow} C[Q]$ since $P \approx_{\Downarrow} Q$, because $\approx_{\Downarrow}$ is a congruence relation, and $C[Q](\bar{l}.n.0) \Downarrow n.Q'$. Hence by lemma 6.2 $C[Q] \Downarrow$ as required.

For all $C[-]$, if $C[P](n.0) \Downarrow n.P'$ then $C[Q](n.0) \Downarrow n.Q'$ and $P' \approx_{\Downarrow} Q'$ (again using congruence of $\approx_{\Downarrow}$). Hence $(P', Q') \in \mathcal{R}$ as required.

The symmetric cases work similarly. Hence $\mathcal{R} \subseteq \{\mathcal{R}\}_b$ and so

$$P \approx_{\Downarrow} Q \Rightarrow P \approx^b_{\Downarrow} Q$$

as required. □

Lemma 6.4

$$P \approx^b_{\Downarrow} Q \Rightarrow P \approx_{\Downarrow} Q$$

Proof

Let $\mathcal{R} = \{(P,Q) \in \text{NCCS} \times \text{NCCS} \mid P \approx^b_{\Downarrow} Q\}$. Then for each $(P,Q) \in \mathcal{R}$ we again deal with the difference cases:

$P(n.0) \Downarrow n.P'$ with $n \notin \text{fn}(P, P', Q)$.

So $Q(n.0) \Downarrow n.Q'$ and $P' \approx^b_{\Downarrow} Q'$.

$P(a.n.0) \Downarrow n.P'$ with $n \notin \text{fn}(P, P', Q)$.

So $P(a.n.0)(\overline{n}.n'.0)(n'.0) \Downarrow n'.(P'(n'.0))$ with $n' \notin \text{fn}(P, P', Q)$. Hence $Q(a.n.0)(\overline{n}.n'.0)(n'.0) \Downarrow n'.(Q'')$ and $P'(n'.0) \approx^b_{\Downarrow} Q''$. We now consider what Q'' may be. Since n' is a new name there must be exactly one

occurrence of n' in Q'', so it can only be one of the following three options:

$$\begin{array}{rcll} Q'' & \equiv & Q'(a.n.0)(\overline{n}.n'.0) & (1) \\ & \text{or} & Q'(n.0)(\overline{n}.n'.0) & (2) \\ & \text{or} & Q'(n'.0) & (3) \end{array}$$

We let $\vec{m} = \text{fn}(P', Q)$. Then we observe that $\nu\vec{m}na.(P'(n'.0)) \Downarrow$ but $\nu\vec{m}n'a.(P'(n'.0)) \not\Downarrow$. Therefore since $P' \approx^b_{\Downarrow} Q''$ we must also have $\nu\vec{m}na.Q'' \Downarrow$ and $\nu\vec{m}n'a.Q'' \not\Downarrow$. If Q'' is structurally equivalent to the form (1) then $\nu\vec{m}na.(Q'(a.n.0)(\overline{n}, n'.0)) \Downarrow$ implies that for some Q''' we have $\nu\vec{m}na.(Q'(a.n.0)(\overline{n}, n'.0)) \Downarrow n'.Q'''$ since n' is the only free name. By corollary 2.17 on page 31 we have $\nu\vec{m}a.(Q'(a.n.0)(\overline{n}, n'.0)) \Downarrow n'.R$ for some R and hence lemma 6.2 implies that $\nu\vec{m}a.(Q'(a.n.0)) \Downarrow n.R'$ for some R'. This in turn means that $\nu\vec{m}a.(Q'(a.n.0)(\overline{n}, n'.0)) \Downarrow n'.(R'(\overline{n}, n'.0))$ by lemma 2.14 on page 29. But this is a contradiction. So Q'' cannot satisfy (1). Similarly it cannot satisfy (2). Therefore $Q'' \equiv Q'(n'.0) \approx^b_{\Downarrow} P'(n'.0)$. Hence we know that

$$Q(a.n.0)(\overline{n}.n'.0)(n'.0) \Downarrow n'.(Q'(n'.0))$$

Therefore by theorem 5.5(1) we have R and i with

$$Q(a.n.0)(\overline{n}.n'.0)(n'.0) \xrightarrow{\tau^i n'} R|0|0|n'.0$$

and $R|0|0|n'.0 \sim Q'|n'.0$[1] (again since $\rightarrow$ is structure preserving and n, n' are not free in Q). We let $R_0 = Q(a.n.0)(\overline{n}.n'.0)(n'.0)$, $R_{i+1} = R|0|0|n'.0$ and the other R_j be the intermediate values. There is is a j_1 with R_{j_1+1} able to input on channel n but an input on channel n not possible for R_{j_1}. Then

$$R_{j_1} = R'_{j_1}(a.n.0)(\overline{n}.n'.0)(n'.0)$$
$$R_{j_1+1} = R'_{j_1+1}(n.0)(\overline{n}.n'.0)(n'.0)$$

Similarly there is a j_2 with R_{j_2} able to input on channel n but R_{j_2+1} cannot. Then

$$R_{j_2} = R'_{j_2}(n.0)(\overline{n}.n'.0)(n'.0)$$
$$R_{j_2+1} = R'_{j_2}(n'.0)(n'.0)$$

[1] Actually we could have $R|0|n'.0|0$ but this won't make any difference. Hence we only follow one of the two possibilities through.

By examining the rule of the transition relation we may see that if $P|Q \xrightarrow{\alpha} P'Q$ then we also know that $P \xrightarrow{\alpha} P'$. Therefore

$$Q(a.n.0) \xrightarrow{\tau^{j_1+1}} R'_{j_1+1}(n.0)$$
$$R'_{j_1+1}(n.0) \xrightarrow{\tau^{j_2-j_1-1}} R'_{j_2}(n.0)$$
$$R'_{j_2}(n.0) \xrightarrow{\tau^{i-j_2-1}} R(n.0)$$

Therefore $Q(a.n.0) \xrightarrow{\tau^* n} R$. However we know that $\sim$ is a congruence relation so $R \sim \nu n'.(R|0|0|n'.0) \sim \nu n'.(Q'|n'.0) \sim Q'$. Then theorem 5.5(2b) shows that there is a Q''' with $Q(a.n.0) \Downarrow n.Q'''$ and $R \sim Q'''$. Hence $Q' \sim Q'''$ which in turn means that $Q' \approx^b_{\Downarrow} Q'''$ by lemmas 6.1 and 6.3 and theorem 3.11.

We also note that since $\approx^b_{\Downarrow}$ is a congruence relation we have $\nu n'.(Q'(n'.0)) \approx^b_{\Downarrow} \nu n'.(P'(n'.0))$. But $\nu n'.n'.0 \approx^b_{\Downarrow} 0$ and so we have $P' \approx^b_{\Downarrow} Q'$. The transitivity of $\approx^b_{\Downarrow}$ gives $P' \approx^b_{\Downarrow} Q'''$ as required.

The symmetric cases follow similarly. Therefore $\mathcal{R} \subseteq \{\mathcal{R}\}_w$ and hence the result follows. □

Proof of 3.14

This follows directly from lemmas 6.3 and 6.4. The second part can be seen from looking at the proof of lemma 6.4 and noting that we only use these cases. □

6.4 Equivalence of evaluation and delay bisimulation

We prove the equivalence of delay bisimulation and evaluation bisimulation. In doing so, we will also prove that we may restrict the parallel contexts for evaluation bisimulation as described in definition 3.6 on page 42. We proceed by showing a cycle of inclusions. It is obvious that the $\simeq_{\Downarrow}$ is included in $\simeq'_{\Downarrow}$ since we cover all the required cases. We go on to prove firstly that restricted evaluation bisimulation is included in delay bisimulation and then we complete the cycle by showing that delay bisimulation is included in evaluation bisimulation.

First we prove a couple of useful lemmas:

Lemma 6.5

$\forall R \;\; P \simeq^d_{\rightarrow} Q \Rightarrow PR \simeq^d_{\rightarrow} QR$

Proof

We follow a very similar method to that used in [Mil89] for proving the equivalent result for strong equivalence. Let $\mathcal{S} = \{(PR, QR) \text{ s.t. } P \simeq^d_{\rightarrow} Q\}$. If $(PR, QR) \in \mathcal{S}$ and $PR \xrightarrow{\alpha} T$ then there are three cases to consider.

1. $P \xrightarrow{\alpha} P'$ and $T \equiv P'R$.

   Then because $P \simeq^d_{\rightarrow} Q$ we have $Q \xrightarrow{\tau^*\alpha} Q'$ with $P' \simeq^d_{\rightarrow} Q'$. Hence $QR \xrightarrow{\tau^*} Q'R$ and $(P'R, Q'R) \in \mathcal{S}$.

2. $R \xrightarrow{\alpha} R'$ and $T \equiv PR'$.

   Then $QR \xrightarrow{\alpha} QR'$ and $(PR', QR') \in \mathcal{S}$.

3. $\alpha = \tau, P \xrightarrow{l} P', R \xrightarrow{\bar{l}} Q'$ and $T \equiv P'R'$

   Then because $P \simeq^d_{\rightarrow} Q$ we have $Q \xrightarrow{\tau^* l} Q'$ with $P' \simeq^d_{\rightarrow} Q'$. And so we get $QR \xrightarrow{\tau^*} Q'R'$ and $(P'R', Q'R') \in \mathcal{S}$.

Using a symmetric argument the result follows. □

Lemma 6.6

For all P, Q then $P \simeq'_{\Downarrow} Q$ implies that $\nu n.P \simeq'_{\Downarrow} \nu n.Q$ for any channel name n.

Proof

We let $\mathcal{R} = \{(P, Q) \text{ s.t. } P \equiv \nu n.P', Q \equiv \nu n.Q' \;\; \& \;\; P' \simeq'_{\Downarrow} Q'\}$. So if $\nu n.P \mathcal{R} \nu n.Q$ and $\nu n.P|x.0 \Downarrow l.P'$ with x fresh in $\nu n.P$ (and hence in particular different from n), by corollary 2.17 on page 31, there is a P'' with $P|x.0 \Downarrow l.P''$ and $\nu n.P'' \equiv P'$. But $P \simeq'_{\Downarrow} Q$ and so we also have Q'' with $Q|x.0 \Downarrow l.Q''$ and $P'' \simeq'_{\Downarrow} Q''$. Then by lemma 2.14 on page 29 $\nu n.Q|x, 0 \Downarrow l.\nu n.Q''$ (since l is not n) and $P' \mathcal{R} \nu n.Q''$.

The symmetric case follows identically. Therefore $\mathcal{R} \subseteq \simeq'_{\Downarrow}$ which gives the result. □

Lemma 6.7

For any NCCS processes P and Q $\;\; P \simeq'_{\Downarrow} Q$ implies $P|\nu n.n.0 \simeq'_{\Downarrow} Q$

Proof

We first observe that $\nu n.n.0 \sim 0$. Then by theorem 3.17 we may derive that $P^T|\nu n.n.0 \sim P^T|0 \sim P^T$. Then lemma 6.1 implies that $P|\nu n.n.0 \simeq'_{\Downarrow} P$. By transitivity of $\simeq'_{\Downarrow}$ we get $P|\nu n.n.0 \simeq'_{\Downarrow} Q$, as required. □

Lemma 6.8

$\forall P, Q, x \notin \mathrm{fn}(P,Q)\ \ P|x.0 \simeq'_{\Downarrow} Q|x.0 \Rightarrow P \simeq'_{\Downarrow} Q$

Proof

By lemma 6.6 $\nu x.(P|x.0) \simeq'_{\Downarrow} \nu x.(Q|x.0)$. Hence

$$\nu x.(P|x.0) \simeq'_{\Downarrow} P|\nu x.x.0 \simeq'_{\Downarrow} P$$

using lemma 6.7 and the fact that $P \equiv Q$ implies that $P \simeq'_{\Downarrow} Q$. Similarly $\nu x.(Q|x.0) \simeq'_{\Downarrow} Q$ and therefore $P \simeq'_{\Downarrow} Q$ as required.

□

Lemma 6.9

$P \simeq'_{\Downarrow} Q \Rightarrow P \simeq^d_{\rightarrow} Q$

Proof

Let $\mathcal{S} = \left\{(P,Q) \text{ s.t. } P \simeq'_{\Downarrow} Q\right\}$. Then if $(P,Q) \in \mathcal{S}$ there are two cases

$P \xrightarrow{\tau} P'$ Then for $x \notin \mathrm{fn}(P,Q)$ there is a P'' with $P|x.0 \Downarrow x.P''$ and $(P')^t \simeq'_{\Downarrow} P''$ (using theorem 5.5(2a) and lemma 6.7). Therefore $Q|x.0 \Downarrow x.Q''$ with $P'' \simeq'_{\Downarrow} Q''$ and there is a Q' s.t. $Q' \sim Q''$ and $Q|x.0 \xrightarrow{\tau^* x} Q'|0$. Hence $Q \xrightarrow{\tau^*} Q'$[2] as desired (and $(P')^t \simeq'_{\Downarrow} Q'$).

$P \xrightarrow{l} P'$ Hence $P \Downarrow l.P''$ for some $P'' \sim P'$. Therefore $P|x.0 \Downarrow l.(P''|x.0)$, for some fresh name x, which implies that $Q|x.0 \Downarrow l.Q''$ with $P''|x.0 \simeq'_{\Downarrow} Q''$. But $Q'' \equiv Q'''|x.0$, for some Q''', (since x is not free in Q) and hence $Q|x.0 \Downarrow l.(Q'''|x.0)$. Using lemma 6.8 gives $P'' \simeq'_{\Downarrow} Q'''$.

Then by theorem 5.5(1) there is a Q'''' with $Q|x.0 \xrightarrow{\tau^* l} Q''''$ and $Q'''' \sim Q'''|x.0$. Again since $\rightarrow$ is structure preserving and x is not free in Q there must be a Q' with $Q'''' = Q'|x.0$. Then $Q \xrightarrow{\tau^* l} Q'$.

[2]This follows easily by induction on the number of τ steps. If there are none then the result may be read off. Otherwise it is obvious from the definition of the transition rules that if $Q_i|x.0 \xrightarrow{\tau} Q'_{i+1}$ then $Q'_{i+1} = Q_{i+1}|x.0$.

But $Q' \sim \nu x.(Q'|x.0) \sim \nu x.(Q'''|x.0) \sim Q''' \simeq'_{\Downarrow} P''$, therefore $P'' \simeq'_{\Downarrow} Q'$ as required.

Using a symmetric argument we see that $\mathcal{S} \subseteq [\mathcal{S}]_d$ and so $\mathcal{S} \subseteq \simeq^d_{\rightarrow}$. Therefore $P \simeq'_{\Downarrow} Q$ implies that $P \simeq^d_{\rightarrow} Q$, as required.

□

Lemma 6.10

$P \simeq^d_{\rightarrow} Q \Rightarrow P \simeq_{\Downarrow} Q$

Proof

Let $\mathcal{S} = \{(P,Q) \text{ s.t. } P \simeq^d_{\rightarrow} Q\}$. Then if $(P,Q) \in \mathcal{S}$ and $PR \Downarrow \alpha.P'$ then, by lemma 6.5, $PR \simeq^d_{\rightarrow} QR$. Hence we have $PR \xrightarrow{\tau^*\alpha} P''$ with $P' \sim P''$ and so $QR \xrightarrow{\tau^*\alpha} Q''$ with $P'' \simeq^d_{\rightarrow} Q''$. Therefore by theorem 5.5(2b) there is a Q' with $Q'' \sim Q'$ and $QR \Downarrow \alpha.Q'$. Noting that $P' \sim P'' \simeq^d_{\rightarrow} Q'' \sim Q'$ implies that $P' \simeq^d_{\rightarrow} Q'$ gives the desired result. A symmetric argument completes the proof. □

Restatement of Theorem 3.7

1. $\simeq^o_{\Downarrow}$ is an equivalence relation.
2. $\simeq^o_{\Downarrow}$ is a congruence relation.
3. $\simeq_{\Downarrow}$ coincides with $\simeq'_{\Downarrow}$

Restatement of Theorem 3.24

1. $\simeq^d_{\rightarrow}$ is a congruence relation.
2. $\simeq^d_{\rightarrow}$ coincides with $\simeq_{\Downarrow}$

Proof of 3.7(3) and 3.24

Observe that $P \simeq_{\Downarrow} Q \Rightarrow P \simeq'_{\Downarrow} Q$. Then lemmas 6.9 and 6.10 show that

$$\simeq_{\Downarrow} \subseteq \simeq'_{\Downarrow} \subseteq \simeq^d_{\rightarrow} \subseteq \simeq_{\Downarrow}$$

The results then follow easily using theorem 3.7(2) and theorem 2.27 for the congruence part of the proof. □

6.5 Inequalities between bisimilarities

We have now shown all the inclusions and equalities of relationships given at the end of chapter 3. We still however need to show that the bisimilarities are not all equal. The following examples show that the bisimilarities do indeed split into three groups. Firstly we observe that

$$\nu n.(\overline{n}.0|n.a.0) \not\sim a.0$$

but by considering the definition of delay bisimulation (definition 3.22 on page 49) we may see that

$$\nu n.(\overline{n}.0|n.a.0) \simeq a.0$$

The remaining inequality is shown by considering delay bisimulation and observing that if

$$P = a.(\nu n.(\overline{n}.0|(b.0 + n.c.0))) + a.c.0 \tag{6.6}$$

$$Q = a.(\nu n.(\overline{n}.0|(b.0 + n.c.0))) \tag{6.7}$$

then

$$P \xrightarrow{a} c.0 = P' \tag{6.8}$$

but then the only thing that Q can do to match this is to use the following transition:

$$Q \xrightarrow{a} \nu n.(\overline{n}.0|(b.0 + n.c.0) = Q' \tag{6.9}$$

But now

$$Q' \xrightarrow{b} \nu n.(\overline{n}.0|0) \tag{6.10}$$

and P' cannot match this. Therefore $P \not\simeq^d_{\rightarrow} Q$.

By noticing that the left hand side is essentially just $a.(b.0 + \tau.c.0) + a.c.0$ we may see that the following is true (also from section 7.4 of [Mil89] which gives the axioms, of which this is a special case, for finite processes):

$$a.(\nu n.(\overline{n}.0|(b.0 + n.c.0))) + a.c.0 \approx a.(\nu n.(\overline{n}.0|(b.0 + n.c.0)))$$

Chapter 7

Equational Theory

We let the set of *finite processes,* called the *finite fragment* of NCCS, be the subset of NCCS that does not involve rec terms. Notice that all these processes have finite traces, but not all processes that have finite traces are finite processes. For example $a.\mathbf{rec}\ X.(b.\nu n.(n.X))$ has a finite trace.

We present an equational theory for Evaluation Bisimulation on the finite fragment of NCCS. We do not present here a theory for Weak Bisimulation. We may simply translate the equational theory for weak equivalence given in [Mil89].

We give the equational theory in terms of τNCCS since this makes the presentation clearer. We could use the technique of using saturated normal forms, as in [Mil89], but instead we present a different method that may be a useful alternative in some situations. This method uses a two stage process for defining the equational theory. The first of these is a conversion of processes to a *standard* form. The second of these gives a set of equations over processes in standard form, which we will show to be equivalent to Evaluation bisimulation over processes in standard form.

7.1 Standard forms

Definition 7.1

We define the *standard* form on finite processes as follows:

$$N \ ::= \ \sum \alpha_i.N_i$$

Note 7.2

First observe that 0 is a process in standard form, by taking an empty sum. Secondly note that we automatically have associativity and commutativity of $+$ due to using the summation notation. We tend to write processes in standard form as K, L, M or N.

7.2 Equational Laws

We split the equations that we use to convert a process to standard form into two sets. The first of these are those that mirror the structural congruence relation on NCCS. The second set allow us to expand parallel contexts into a single large sum.

$$
\begin{array}{rcll}
P_1(P_2P_3) & \cong_s & (P_1P_2)P_3 & (S1) \\
P_1P_2 & \cong_s & P_2P_1 & (S2) \\
P0 & \cong_s & P & (S3) \\
\nu n.(P_1P_2) & \cong_s & (\nu n.P_1)P_2 \quad n \notin \mathrm{fn}(P_2) & (S4) \\
\nu n_1.\nu n_2.P & \cong_s & \nu n_2.\nu n_1.P & (S5) \\
\nu n.0 & \cong_s & 0 & (S6)
\end{array}
$$

Figure 7.1: Structural Equations

The equations dealing with the structural congruence are given in figure 7.1. We might be surprised that some rules seem to be missing. We have no rules for $+$, but these are included in the definition of standard forms as commented on in note 7.2.

Lemma 7.3

$P \cong_s Q \Rightarrow P \simeq Q$

Proof

Observe that all the structural equations are instances of the structural congruence, and that two processes that are structurally congruent are evaluation bisimilar and hence also delay bisimilar. □

Figure 7.2 on the next page gives the equations that allow us to convert a general finite process into one in standard form.

$$\begin{aligned}
\nu n.\sum_i \alpha_i.P_i &\cong_e \sum_{i:\alpha_i \notin \{n,\overline{n}\}} \alpha_i.\nu n.P_i && (E1)\\
\sum \alpha_i.P_i \Big| \sum \beta_j.Q_j &\cong_e \sum_i \alpha_i.\left(P_i \Big| \sum \beta_j.Q_j\right) \\
&\quad + \sum_j \beta_j.\left(\sum \alpha_i.P_i \Big| Q_j\right) && (E2)\\
&\quad + \sum_{\alpha_i = \overline{\beta_j}} \tau.\left(P_i \Big| Q_j\right)
\end{aligned}$$

Figure 7.2: Expansion Equations

Lemma 7.4

$P \cong_e Q \Rightarrow P \simeq Q$

Proof

Observe that (E2) is just the expansion law from CCS. Thus we know that the LHS is Strong CCS Equivalent to the RHS and hence they are Evaluation Bisimilar by lemma 6.1 on page 73. It is easy to see from the definition of Strong CCS Equivalence (definition 3.15 on page 47) that if $P \cong_{E1} Q$[1] then $P \sim Q$. The result then follows by lemma 6.1. □

We have one further set of equations. They first deal with summation and the τ-laws. They are given in figure 7.3. While these rules are true for general processes, we will only apply them to processes in standard form.

$$\begin{aligned}
N &\cong_p N + 0 && (P1)\\
N + N' &\cong_p N + N + N' && (P2)\\
N &\cong_t \tau.N && (T1)\\
\tau.N + N' &\cong_t \tau.N + N + N' && (T2)
\end{aligned}$$

Figure 7.3: Summation and τ Laws

We do not need rule (P1) since it is contained within the definition of summation, however we include it for clarity. The presence of N' in (P2) and (T2) may seem a little surprising. It is there because we will later restrict the contexts in which the various equations may be used. Therefore to be able to

[1] $P \cong_{E1} Q$ if $P = \nu n.\sum_i \alpha_i.P_i$ for some n, α_i and P_i and $Q = \sum_{i:\alpha_i \notin \{n,\overline{n}\}} \alpha_i.\nu n.P_i$.

allow (P2) and (T2) to be used in any sum we must have the N' term there. Another possible problem is how to show that, for example $N = N + N$. This will follow using (P1) to get $N + 0$, then (P2) to get $N + N + 0$, followed by (P1) to get $N + N$ as required. We might be surprised that there is not an N' term in (P1) and (T1). In the (P1) case this is because it is not needed. We just view N as "$N + N'$". For (T1) we do not have N' since the result is no longer true with it there. Notice that $a.0 \simeq \tau.a.0$ (from the definition of delay bisimulation) but $a.0 + b.0 \not\simeq \tau.a.0 + b.0$, since $\tau.a.0 + b.0 \xrightarrow{\tau} a.0$ while $a.0 + b.0$ cannot do anything equivalent and end up bisimilar to $a.0$.

Lemma 7.5

1. $P \cong_p Q \Rightarrow P \simeq Q$
2. $P \cong_t Q \Rightarrow P \simeq Q$

Proof

Observe that (P1) and (P2) are both contained within Strong CCS Equivalence and hence are in Evaluation Bisimulation. (T1) and (T2) both follow easily from the definition of delay bisimulation (definition 3.22 on page 49). □

Definition 7.6

We let $\cong_{[sept]}$ be the union of $\cong_s$, $\cong_e$, $\cong_p$ and $\cong_t$.

We can now define the equational theory for the finite fragment of τNCCS.

$$\frac{P \cong_{[sept]} Q}{C[P/X] \cong C[Q/X]} \quad C[X] \text{ open } \tau\text{NCCS process}$$

$$\frac{P \cong Q \quad Q \cong R}{P \cong R} \qquad \frac{P \cong Q}{Q \cong P}$$

Figure 7.4: Equational Theory for Evaluation Bisimulation

Definition 7.7

For closed τNCCS processes we let $\cong$ be defined as in figure 7.4.

It may look as though our definition is overly restrictive in only allowing one substitution of a process at a time. Because we have transitivity this

does not matter. We can encode a polyadic version in terms of the above definition. We do not define $\cong$ over processes with free variables. The only way to bind a free variable is by using a rec term. However we may not have a rec term in a finite process so this case cannot occur. We therefore would not add anything by allowing free variables.

Lemma 7.8

For P, Q in the finite fragment of τNCCS $P \cong Q \Rightarrow P \simeq Q$

Proof

We know that $\simeq$ is a congruence relation. The result then follows from lemmas 7.3, 7.4 and 7.5. □

7.3 Characterization of Evaluation Bisimulation

In order to show that the equational theory characterizes Evaluation Bisimulation on Finite Processes we now proceed by a two step argument. We first show that any Finite Process is equivalent, in the equational theory, to one in Standard form. We will then show that Evaluation Bisimularity on Standard Forms implies that the two processes are equivalent in the equational theory.

Lemma 7.9

For any Finite Process P there is a process, N, in standard form with $P \cong N$.

Proof

We use the following three step method (noting that we use α-conversion without comment):

1. Use (S2), (S4), (S5), (S6) and (E1) to pull out all restrictions.

2. Use (S3) and (E2) to expand out all parallel contexts into a single sum.

3. Use (S6) and (E1) to push all restrictions in, discarding restrictions that do not restrict any channels and removing elements of sums that are restricted out.

□

Note 7.10

We do not use the rule (S1) in lemma 7.9. This is principally because it does not matter in what order parallel composition is expanded out since summation is commutative.

We need to define the depth of a process in Standard Form that we will use in various proofs.

Definition 7.11

We define the depth $d(N)$ of a standard form as:

$$\begin{aligned} d(0) &= 0 \\ d(\alpha.N) &= 1 + d(N) \\ d(\textstyle\sum N_i) &= \max(d(N_i)) \end{aligned}$$

We now prove various lemmas which we will use later.

Lemma 7.12

For any process, N, in standard form $\mathrm{fn}(N) = \emptyset$ implies that $N \cong 0$.

Proof

We prove this by induction of the depth of N. If $d(N) = 0$ then we are done. Otherwise $N = \sum \tau.N_i$ since it must be $\sum \alpha_i.N_i$ and α_i must be τ. Then by induction hypothesis each $N_i \cong 0$ and hence $N \cong \sum_i \tau.0$. Then the result follows using (P1), (P2) and (T1). □

Lemma 7.13

For processes N, N' and M in standard form then

1. $\tau.N \xrightarrow{\tau^*} N'$ implies that $\tau.N + M \cong \tau.N + N' + M$.
2. $\tau.N \xrightarrow{\tau^*\alpha} N'$ implies that $\tau.N + M \cong \tau.N + \alpha.N' + M$.

Proof

These both follow from using induction on the number of τ steps and (T2) multiple times, or (P2) is there are no τ steps. In particular if $N \stackrel{\tau^i}{\rightarrow} N'$ and

$$N = N_0 \stackrel{\tau}{\rightarrow} N_1 \ldots \stackrel{\tau}{\rightarrow} N_i = N'$$

then

$i = 1$ The result follows immediately using (T2).

$i > 1$ Then $N_1 \stackrel{\tau}{\rightarrow} N_2$ and $N_1 = \tau.N_2 + N_1'$. Using the induction hypothesis on $\tau.N_2 + N + N_1' + M$ gives the result.

The second part follows identically. □

Note 7.14

We write τ^+ as shorthand for $\tau\tau^*$. Thus it denotes one or more τ's.

Corollary 7.15

For processes N, N', M and M' in standard form with $N' \cong M'$ then

1. $N \stackrel{\tau^+}{\rightarrow} N'$ implies that $N + M \cong N + M' + M$.
2. $N \stackrel{\tau^*\alpha}{\longrightarrow} N'$ implies that $N + M \cong N + \alpha.M' + M$.

Proof

We let $N = \sum \alpha_i.N_i$. If there are no τ steps, which can only occur for part (2), then $N' = \alpha_i.N_i$ for some i. The result then follows using (P2) and the fact that $N + \alpha.N' + M \cong N + \alpha.M' + M$. Otherwise there is an i with $\alpha_i = \tau$ and $\tau.N_i \stackrel{\tau^*}{\rightarrow} \tau.N'$ (or $\tau.N_i \stackrel{\tau^*}{\rightarrow} \alpha.N'$ for part (2)). Then by lemma 7.13 $N + M \cong N + \tau.N' + M$ (or $N + M \cong N + \alpha.N' + M$ for part (2)). Therefore $N + M \cong N + \tau.M' + M$ (or $N + M \cong N + \alpha.M' + M$, as required, for part (2)). Hence $N + M \cong N + M' + M$, as required, using (T2) and $N + M \cong N + \tau.N' + M$.

□

We now finish proving theorem 7.17. We proceed by a rather complicated induction proof, in which we use the properties of delay bisimulation without comment (see definition 3.22 on page 49). If $M \simeq N$ then we assume,

without loss of generality, that $d(M) \geq d(N)$. The induction proceeds on the lexicographic ordering on $(d(M), d(N))$. This may look like a rather strange ordering, but the proof proceeds by showing that we can transform M into N using only the equations given in figure 7.3 on page 87 and figure 7.4 on page 88. The main idea, used repeatedly throughout the proof, is that if $M \xrightarrow{\tau} M'$ then there must be a N' such that $N \xrightarrow{\tau^*} N'$ with $M' \simeq N'$. But $d(M') < d(M)$ so by the induction hypothesis $M' \cong N'$. Using this (possibly multiple times) will often allow us to reduce the lexicographic depth. We also use a similar idea with $M \xrightarrow{\alpha} M'$. We find that the proof proceeds easily if $d(M) \neq d(N)$ so we only comment further on the case when $d(M) = d(N)$. In this case we cannot necessarily reduce (in the lexicographic ordering) $(d(M), d(N))$. If we cannot easily reduce the lexicographic depth then we show directly that $M \cong M + N$ (and similarly that $M + N \cong N$ using symmetry). From this we may deduce that $M \cong N$.

Lemma 7.16

For P, Q in the finite fragment of τNCCS $P \simeq Q \Rightarrow P \cong Q$

Proof

We proceed by induction on the lexicographic ordering on $(d(M), d(N))$. We assume w.l.o.g. that $d(M) \geq d(N)$. Then, possibly using (P1), we let

$$\begin{aligned} M &= \sum_{\alpha_i \neq \tau} \alpha_i.M_{1i} + \sum \tau.M_{2j} \\ N &= \sum_{\alpha_k \neq \tau} \alpha_k.N_{1k} + \sum \tau.N_{2l} \end{aligned}$$

We now split into four cases.

$d(N) = 0$

So we have $N = 0$ and hence $N \not\rightarrow$. Therefore we cannot have $M \xrightarrow{\tau^*\alpha}$ with $\alpha \neq \tau$ and hence (because M is in standard form) $\mathrm{fn}(M) = \emptyset$. Therefore by lemma 7.12 $M \cong 0$.

$d(M) > 1 + d(N)$

Since $M \simeq N$ and $M \xrightarrow{\tau} M_{2j}$ for each j, we get $N \xrightarrow{\tau^*} N'_{2j} \simeq M_{2j}$. Similarly we get $N'_{1i} \simeq M_{1i}$ for each i. By induction hypothesis $M_{ij} \cong N'_{ij}$ and hence $M \cong M' = \sum \alpha_i.N'_{1i} + \sum \tau.N'_{2j}$ and $d(M') < d(M)$. Hence $M' \cong N$ by induction hypothesis, giving $M \cong N$ as required.

$d(M) = 1 + d(N)$

The same argument works as in the previous case unless there is a j s.t. $d(N'_{2j}) = d(M) - 1$[2] in which case $N'_{2j} = N$ and $M_{2j} \simeq N$. This implies that $M_{2j} \simeq M$ and by induction hypothesis $M_{2j} \cong N$. Then $M \cong M_{2j}$ by using corollary 7.15 once for each $M_{2j'}$, $j' \neq j$ and once for each M_{1i}. Therefore $M \cong N$ as required.

$d(M) = d(N)$

If there is a j s.t. $M \xrightarrow{\tau} M_{2j}$ with $M_{2j} \simeq N$ then we are done since $d(M_{2j}) < d(M)$ and so $M_{2j} \cong N$ and $M \cong M_{2j}$ using the same reasoning as the $[d(M) > 1 + d(N)]$ case. Similarly if there is a N_{2l} with $N \xrightarrow{\tau} N_{2l}$ and $M \simeq N_{2l}$ we are also done. Otherwise we show that there is a M' such that $M \cong M' \cong N$.

We let M' be $M + N$. So

$$M' = \sum_{\alpha_i \neq \tau} \alpha_i.M_{1i} + \sum \tau.M_{2j} + \sum_{\alpha_k \neq \tau} \alpha_k.N_{1k} + \sum \tau.N_{2l}$$

We want to show that $M \cong M'$. We let $M'_0 = M$, $M'_n = M'$ and M'_i has one more term in the sum than M'_{i-1}. Then:

$i = 0$

$M \cong M'_0$.

$0 < i \leq n$

So $M'_i = M'_{i-1} + \alpha.L$, where α may be τ. If $\alpha = \tau$ then $N \xrightarrow{\tau} L$ and so there is a K s.t. $M \xrightarrow{\tau^+} K$[3] with $K \simeq L$ and hence by outer induction hypothesis $K \cong L$. Therefore we also have $M'_{i-1} \xrightarrow{\tau^+} K$. Hence by corollary 7.15 $M'_{i-1} \cong M'_{i-1} + \alpha.L$, since $K \cong L$. Hence $M'_{i-1} \cong M'_{i-1} + \alpha.L$ as required.

Otherwise we have $\alpha \neq \tau$. Then we have $N \xrightarrow{\alpha} L$ and so we also have a K s.t. $M \xrightarrow{\tau^j\alpha} K$, for some j and $K \simeq L$. So we also have $M'_{i-1} \xrightarrow{\tau^j\alpha} K$ and also by outer induction hypothesis we have

[2]Notice that $d(N'_{1i}) < d(N)$ for all i.

[3]We know that we must have at least one τ here since otherwise we come back to the case described above which we have already dealt with.

$K \cong L$. Therefore by corollary 7.15 $M'_{i-1} \cong M'_{i-1} + L = M'_i$, as required.

The same argument, but taking $M'_0 = N$, shows that $M' \cong N$. Therefore $M \cong N$ as required.

□

Theorem 7.17

For P, Q in the finite fragment of τNCCS $P \cong Q \Leftrightarrow P \simeq Q$

Proof

This follows from lemmas 7.8 and 7.16. □

By Theorem 7.17 we have an equational characterization of Delay Bisimulation, which is sound and complete. We can also get a characterisation of Evaluation Bisimulation on NCCS by using the translation given in Definition 2.22 on page 35.

Part II

$\mathbf{CML}_{\mathcal{V}}$

Chapter 8

The Calculus for CML

In this chapter we first present the background of CML. We then go on to give the syntax and operational semantics of CML_ν.

8.1 Introduction to CML

Concurrent ML (CML) is a fully concurrent higher order functional language. It has all the properties normally associated with higher order functional languages. In particular it has higher order recursive functions, and values include not just integers and booleans, but also functions. The functions are evaluated using a call-by-value semantics. The higher order function constructs are augmented with primitives for concurrency. These include the spawning of new processes and the ability of processes to communicate via, possibly dynamically created, channels. These dynamically created channels may also be viewed as private channels, over which only those to whom the channel name has been divulged may communicate. The language is further extended by adding delayed computations. These are also values and may be communicated between processes, along with any other values.

It has already been shown in various contexts, e.g. [Mil90, San93a], that the ability to communicate private channel names, combined with dynamic name creation and recursion, together give enough power to encode both higher order languages and object oriented languages. All these elements are present in CML. However as Sangiorgi mentions in [San93a], while a first-order paradigm should be taken as basic, a higher-order calculus can be very

useful for reasoning at a more abstract level.

Reppy suggests in [Rep92] that there are three important aspects to good language design. The first two are that there should be a real problem that needs solving, and that the language should solve it. The third is that there needs to be a firm theoretical foundation for the design. He also requires that the design should be demonstrated in practice. He gives various problems for which a concurrent, rather than just parallel, language would be useful and, having implemented CML, he constructs a multi-threaded X window system toolkit (eXene). He also shows that CML compares favorably with μSystem, a C light-weight process library. He thus shows that his implementation of CML is a practical language to consider working in. He has also shown that CML solves some real problems. He gives an operational semantics for λ_{cv}. This is a language which embodies the main concurrency features of CML. The operational semantics requires the use of two reduction relations. The first is a polymorphically typed version of Plotkin's λ_v calculus. The second, which extends λ_v to λ_{cv}, deals with the concurrent aspects of the language. The overall reduction relation is based on whole programs and not on program fragments. This means, in particular, that the semantics is not compositional. He also does not consider equivalences between processes. These would be useful for ensuring the validity of possible optimisations that may be used.

In [FHJ95] Ferreira, Hennessy and Jeffrey give a calculus, μCML, which contains the core concepts in CML. They give an operational semantics which is compositional and furthermore uses one unified transition relation. Having defined μCML they then give various equivalence relations on processes. However the calculus does not allow dynamic creation of channel names.

8.2 Syntax of CML$_\nu$

We base CML$_\nu$ on μCML$^+$ given in [FH95] with some small variations. The syntax of CML$_\nu$ is given in figure 8.1 on the opposite page. The main differences between μCML$^+$ and CML$_\nu$ are that CML$_\nu$ has dynamic name

$Exp,\ e$	::=	v	Value
	|	$\mathbf{c}e$	Constant application
	|	**if** e **then** e **else** e	Conditional
	|	(e, e)	Pairing
	|	**let** $x = e$ **in** e	Local declaration
	|	ee	Function application
	|	g	Guarded expression
	|	$e \mid e$	Parallel composition
	|	$\nu n.e$	Channel restriction
$Val,\ v$	::=	ℓ	Literal
	|	**fix** $(x = \mathbf{fn} y \Rightarrow e)$	Recursive functions
	|	$< v, v >$	Evaluated pair
	|	$[g]$	Delayed expression
	|	x	Variable
$GExp,\ g$	::=	$v!v$	Output
	|	$v?$	Input
	|	$g \Rightarrow v$	Wrapper
	|	$g \oplus g$	Binary summation
	|	Λ	Null process
	|	$\nu n.g$	Channel restriction
$Lit,\ \ell$	::=	**true**	True
	|	**false**	False
	|	n	Channel names
	|	$()$	Unit value
	|	i	Integers, $i \in \mathbb{N}$

Figure 8.1: Syntax of CML$_\nu$

creation, using the restriction operator ν, and does not have the always operator A. The former adds to the power of the language, while the lack of the always operator A does not reduce the expressibility of the language. We leave out A so that we may mirror the guarded nature of summation in NCCS.

We describe the syntactic categories bottom up.

Lit The literals are made up from various base types. These include booleans and integers. We also include a countable number of channel names. There is also the unit value which is used to build more complicated functions.

GExp The guarded expressions, written either g or ge, are those which must input or output before reducing to anything else. These correspond to the label guarded summation of NCCS. The null process Λ is included here since we will want to allow summation of a guarded expressions and the null process. It also allows us to deal simply with equivalences, such as Λ being equivalent to $\nu n.n!v$. Channel name restriction is included so that the structural congruence may work freely. In particular we wish to have guarded expressions, such as $\nu n.m!n$, in which we may output a private name. Wrappers are used to allow prefixing. A process may input or output a value on a channel and then continue to behave as the second part of the wrapper. The second part will normally be a function which takes one argument. In the case of input this argument will be the value received. When name restriction is used in conjunction with wrappers we may express mobility. This may be done by generating new channel names and then sending them to other processes.

Val The values are those expressions which are valid results from functions. We also regard them as the set of expressions that may be transmitted over channels. It is easily seen that all literals are reasonable results for functions or for output on a channel. As already mentioned (recursive) functions are regarded as acceptable values, since we may well apply a higher order function to a function. Allowing functions as values also

allows us to have higher-order types. Only pairs in which both elements are values are considered values. Delayed expressions are included in the calculus particularly so that we may communicate processes over channels. In order to do this we wrap up expressions as delayed expressions, which may then be communicated to other processes. These delayed expressions may be evaluated once, many times or never, depending on the process that inputs the delayed expression. Variables are included so that values are replaced by values in function application. This will also apply when a process inputs a value since this will be based on function application.

Exp Expressions may be regarded as being programs, whether whole or fragments. We define our reduction relations on expressions. The categories may be split into two parts, those associated with the aspects of the language to do with higher-order functions, and those associated with the concurrent aspects of the language. Values may be regarded as being in either part, depending on whether they are functions or communications.

The higher order function constructs include function application, both that of constant application and that of applying a (possibly higher order) function to a value. More discussion of constants is included below. There are also primitives for conditional execution of expressions. Pairing is used both for sequencing evaluation of expressions and also for giving results of more than one value.

The constructs used in the concurrent aspect of the language are based mainly on the communication channels. We have guarded expressions which, as described above, are guarded by either an input or an output. There is also name restriction, with the associated concept of dynamic name creation. In addition to these we allow processes to occur in parallel, so that they may communicate with one another.

Figure 8.2 on the next page gives the constant functions which we will use throughout the consideration of CML$_\nu$. The only constant function that is included that was not in μCML$^+$ [FH95] is the match function. We include

this for convenience and because we may need to compare restricted names.

• **fst**	First value of an evaluated pair.
• **snd**	Second value of an evaluated pair.
• **add**	Addition of two integers.
• **mul**	Multiplication of two integers.
• **leq**	Less than or equal to.
• **transmit**	Output a value on a channel.
• **receive**	Input a value on a channel.
• **choose**	Binary choice.
• **spawn**	Create a new process.
• **sync**	Launch a delayed computation.
• **wrap**	Combine delayed computations.
• **never**	Delayed computation which deadlocks.
• **always**	Delayed computation which immediately evaluates to a value.
• **match**	Test for equality of names.

Figure 8.2: Constant functions

We could also include more complicated arithmetic or boolean functions and primitives for channel name creation. However we will concentrate on the core calculus.

Definition 8.1

We define the *free* and *bound* names (and variables) in figure 8.3 on the opposite page.

Definition 8.2

We say that a name, n, is *fresh* in e if it does not appear in e as a free or bound name or variable. We typically do not specify e when it is obvious from the context, and take the most general e possible.

Some of the rules for determining the free names and variables may look unconventional at first sight, in particular the rule for $v_1!v_2$. The reason for this is that v_1 may well be a variable, not a channel name. This may occur, for

Expression	fn	bn	fv	bv
ce	$\mathrm{fn}(e)$	$\mathrm{bn}(e)$	$\mathrm{fv}(e)$	$\mathrm{bv}(e)$
if e_1 **then** e_2 **else** e_3	$\bigcup_i \mathrm{fn}(e_i)$	$\bigcup_i \mathrm{bn}(e_i)$	$\bigcup_i \mathrm{fv}(e_i)$	$\bigcup_i \mathrm{bv}(e_i)$
(e_1, e_2)	$\bigcup_i \mathrm{fn}(e_i)$	$\bigcup_i \mathrm{bn}(e_i)$	$\bigcup_i \mathrm{fv}(e_i)$	$\bigcup_i \mathrm{bv}(e_i)$
let $x = e_1$ **in** e_2	$\bigcup_i \mathrm{fn}(e_i)$	$\bigcup_i \mathrm{bn}(e_i)$	$(\mathrm{fv}(e_2) \setminus \{x\}) \cup \mathrm{fv}(e_1)$	$\bigcup_i \mathrm{bv}(e_i) \cup (\{x\} \cap \mathrm{fv}(e_2))$
$e_1 e_2$	$\bigcup_i \mathrm{fn}(e_i)$	$\bigcup_i \mathrm{bn}(e_i)$	$\bigcup_i \mathrm{fv}(e_i)$	$\bigcup_i \mathrm{bv}(e_i)$
$e_1 \vert e_2$	$\bigcup_i \mathrm{fn}(e_i)$	$\bigcup_i \mathrm{bn}(e_i)$	$\bigcup_i \mathrm{fv}(e_i)$	$\bigcup_i \mathrm{bv}(e_i)$
$\nu n.e$	$\mathrm{fn}(e) \setminus \{n\}$	$\mathrm{bn}(e) \cup (\{n\} \cap \mathrm{fn}(e))$	$\mathrm{fv}(e)$	$\mathrm{bv}(e)$
fix $(x = \mathbf{fn} y \Rightarrow e)$	$\mathrm{fn}(e)$	$\mathrm{bn}(e)$	$\mathrm{fv}(e) \setminus \{x, y\}$	$\mathrm{bn}(e) \cup (\{x, y\} \cap \mathrm{fv}(e))$
$< v_1, v_2 >$	$\bigcup_i \mathrm{fn}(v_i)$	$\bigcup_i \mathrm{bn}(v_i)$	$\bigcup_i \mathrm{fv}(v_i)$	$\bigcup_i \mathrm{bv}(v_i)$
[ge]	$\mathrm{fn}(ge)$	$\mathrm{bn}(ge)$	$\mathrm{fv}(ge)$	$\mathrm{bv}(ge)$
x	$\emptyset$	$\emptyset$	x	$\emptyset$
$v_1 \vert v_2$	$\bigcup_i \mathrm{fn}(v_i)$	$\bigcup_i \mathrm{bn}(v_i)$	$\bigcup_i \mathrm{fv}(v_i)$	$\bigcup_i \mathrm{bv}(v_i)$
$v?$	$\mathrm{fn}(v)$	$\mathrm{bn}(v)$	$\mathrm{fv}(v)$	$\mathrm{bv}(v)$
$ge \Rightarrow v$	$\mathrm{fn}(ge) \cup \mathrm{fn}(v)$	$\mathrm{bn}(ge) \cup \mathrm{bn}(v)$	$\mathrm{fv}(ge) \cup \mathrm{fv}(v)$	$\mathrm{bv}(ge) \cup \mathrm{bv}(v)$
$ge_1 \oplus ge_2$	$\bigcup_i \mathrm{fn}(ge_i)$	$\bigcup_i \mathrm{bn}(ge_i)$	$\bigcup_i \mathrm{fv}(ge_i)$	$\bigcup_i \mathrm{bv}(ge_i)$
Λ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
true, false	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
n	n	$\emptyset$	$\emptyset$	$\emptyset$
$()$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 8.3: Free and bound names and variables

example, in the following type of expression:

$$a? \Rightarrow \mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow y!\mathbf{true})$$

In this case, the channel name over which true should be return is received on channel a.

Note 8.3

When x is not a free variable in e we will write $\mathbf{fn}\,(y \Rightarrow e)$ as shorthand for $\mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow e)$.

Definition 8.4

We call a CML_ν expression a *process* if it does not have a parallel composition at the outer level. We let *closed* CML_ν expressions be the set of CML_ν expressions that have no free variables.

8.3 Structural congruence on CML_ν

As in NCCS we are going to use a structural congruence to simplify the reduction relations.

$$\begin{array}{rcll} \nu n.e & \equiv_\alpha & \nu m.e[m/n] & m \notin \text{fn}(e) \\ \mathbf{let}\ x = e\ \mathbf{in}\ f & \equiv_\alpha & \mathbf{let}\ y = e\ \mathbf{in}\ f[y/x] & y \notin \text{fv}(f) \\ \mathbf{fix}\,(x = \mathbf{fn} z \Rightarrow e) & \equiv_\alpha & \mathbf{fix}\,(y = \mathbf{fn} w \Rightarrow e[y/x, w/z]) & y, w \notin \text{fv}(e) \end{array}$$

Figure 8.4: α-conversion for CML_ν

Definition 8.5

The structural congruence, $\equiv$, is defined to be the smallest congruence containing the rules given in figure 8.5 on the next page, using the definition of α-conversion given in figure 8.4.

The side conditions for α-conversion at first sight may look to be rather over-restrictive. They are required to ensure that only the names or variables bound by the outermost binder may be changed. The over-restrictiveness is rather ephemeral because we may use α-conversion on the inner parts of the expression, before using the required rule, to ensure the side conditions

hold, and then afterwards to return the bound names and variables of the inner expressions to the required values. It is also worth recalling that names and variables are distinct, being of different syntactic types.

$$
\begin{array}{rcll}
e & \equiv & f & e \equiv_\alpha f \\
\\
\Lambda|e & \equiv & e & \\
\nu n.\Lambda & \equiv & \Lambda & \\
(e_0|e_1)|e_2 & \equiv & e_0|(e_1|e_2) & \\
\\
\nu n.\nu m.e & \equiv & \nu m.\nu n.e & \\
\nu n.(e_1|e_2) & \equiv & (\nu n.e_1)|e_2 & n \notin \mathrm{fn}(e_2) \\
\nu n.(e_1|e_2) & \equiv & e_1|\nu n.e_2 & n \notin \mathrm{fn}(e_1) \\
\nu n.(ce) & \equiv & c(\nu n.e) & \\
\nu n.(e_1, e_2) & \equiv & (\nu n.e_1, e_2) & n \notin \mathrm{fn}(e_2) \\
\nu n.(e_1, e_2) & \equiv & (e_1, \nu n.e_2) & n \notin \mathrm{fn}(e_1) \\
\nu n.(e_1 e_2) & \equiv & (\nu n.e_1)e_2 & n \notin \mathrm{fn}(e_2) \\
\nu n.(e_1 e_2) & \equiv & e_1(\nu n.e_2) & n \notin \mathrm{fn}(e_1) \\
\nu n.(\textbf{let } x = e_1 \textbf{ in } e_2) & \equiv & \textbf{let } x = e_1 \textbf{ in } \nu n.e_2 & n \notin \mathrm{fn}(e_1) \\
\nu n.(\textbf{let } x = e_1 \textbf{ in } e_2) & \equiv & \textbf{let } x = \nu n.e_1 \textbf{ in } e_2 & n \notin \mathrm{fn}(e_2) \\
\nu n.(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) & \equiv & \textbf{if } \nu n.e_0 \textbf{ then } e_1 \textbf{ else } e_2 & n \notin \mathrm{fn}(e_1, e_2) \\
\nu n.(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) & \equiv & \textbf{if } e_0 \textbf{ then } \nu n.e_1 \textbf{ else } \nu n.e_2 & n \notin \mathrm{fn}(e_0) \\
\nu n.(g \Rightarrow v) & \equiv & (\nu n.g) \Rightarrow v & n \notin \mathrm{fn}(v) \\
\nu n.(g_1 \oplus g_2) & \equiv & \nu n.g_1 \oplus g_2 & n \notin \mathrm{fn}(g_2) \\
\nu n.(g_1 \oplus g_2) & \equiv & g_1 \oplus \nu n.g_2 & n \notin \mathrm{fn}(g_1)
\end{array}
$$

Figure 8.5: Structural Congruence for CML$_\nu$

The structural equivalence may be split into three parts. The first deals with α-conversion and is given in more detail in figure 8.4 on the opposite page. The second includes the treatment of the dead process Λ. We allow the deletion of the dead process, but only when it is not the rightmost process in the expression! This might look rather peculiar. The rightmost process is special, in that it is the only process that may return a value.

It may be viewed as the controlling or original process, which returns the result of the program to the environment. We also include the notion that restricting a name in the dead process does not do anything, so the restricted and unrestricted versions may be regarded as the same. In this part of the structural congruence we also include the enforcement of associativity on parallel composition.

The third part deals with the scope of name restrictions. Alternatively this may be viewed as an expression of the context which knows about a new dynamically created name. We allow the scope to be changed so long as we do not bind any previously free names, nor free any previously bound names.

Observe that we do not have a rule $\nu n.e \equiv e$ with the side condition that $n \notin \mathrm{fn}(e)$. This is because it is derivable thus (remembering that $n \notin \mathrm{fn}(e)$):

$$e \equiv \Lambda|e \equiv (\nu n.\Lambda)|e \equiv \nu n.(\Lambda|e) \equiv \Lambda|\nu n.e \equiv \nu n.e$$

8.4 Evaluation Relation

As well as using a structural congruence we also employ evaluation contexts. These are essentially the contexts which do not block a reduction. These correspond to the $\nu\vec{n}(-)$ in the ($\Downarrow$COMP) rule in NCCS (see figure 2.2 on page 26). It is merely that for CML_ν we will have more complicated contexts, and use them in more rules.

Definition 8.6

The *evaluation contexts* are given in figure 8.6 on the next page.

Examining the evaluation contexts it is obvious that they do not contain all possible contexts. For example the context $(e, \mathcal{E}[-])$ is not included. This is because we want to ensure evaluation proceeds such that the left-hand part of a pair evaluates before the right-hand part. One might also wonder why we do not have a context such as $(v, \mathcal{E}[-])$. Instead of using this context we follow [FHJ95] and reduce (v, e) to "let $x = e$ in $< v, x >$". Similarly we want to ensure that we evaluate a function value before evaluating the argument. Again rather than including a context $v\mathcal{E}[-]$ we allow ve to reduce to "let $y = e$ in $f[v/x]$" where $v = \mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow f)$. In both cases we only allow these reductions if the resulting expression, possibly in the presence of an

$$
\begin{array}{rcl}
\mathcal{E}[-] & ::= & - \\
& | & \nu n.\mathcal{E}[-] \\
& | & \mathcal{E}[-]|e \\
& | & e|\mathcal{E}[-] \\
& | & c\mathcal{E}[-] \\
& | & \mathcal{E}[-]e \\
& | & (\mathcal{E}[-], e) \\
& | & \textbf{if } \mathcal{E}[-] \textbf{ then } e_t \textbf{ else } e_f \\
& | & \textbf{let } x = \mathcal{E}[-] \textbf{ in } e
\end{array}
$$

Figure 8.6: Evaluation Contexts for CML_ν

evaluation context, can then evaluate to a committed form. The committed forms are given in the next definition.

Definition 8.7

We let α, being any input or output, range over $n!v$ and $n?x$ for any channel name n, value v and variable x. We let l, being any label, range over any α and $\surd v$ for any value v. We then let the committed forms C be of the form $\nu\vec{m}.l.e$ with the restrictions that $\{\vec{m}\} = \emptyset$ if $l = n?x$ and $n \notin \{\vec{m}\}$ if $l = n!v$. These restrictions will be enforced by the evaluation relation.

We now define the "big step" evaluation relation for CML_ν. The rules are divided into five groups. The first, given in figure 8.7 are the axioms.

$$
\text{VAL}\ \frac{}{v \Downarrow \surd v.\Lambda} \qquad \text{OUT}\ \frac{}{n!v \Downarrow n!v.()} \qquad \text{IN}\ \frac{}{n? \Downarrow n?x.x}
$$

Figure 8.7: Evaluation Relation on CML_ν — Axioms

These rules are fairly self-explanatory. The first, (VAL), shows that a value may evaluate to return itself, and then continue as the dead process. The other two rules output or input a value on a channel respectively. (OUT) continues with the unit value, which is a simple value, so that prefixing is possible using wrapping. When a process inputs a value, using (IN), it

continues with the value, which may then be passed to a function using wrapping.

$$\text{SUM}_1 \quad \frac{ge_1 \Downarrow \nu\vec{n}.\alpha.e}{ge_1 \oplus ge_2 \Downarrow \nu\vec{n}.\alpha.e} \quad \{\vec{n}\} \cap \text{fn}(ge_2) = \emptyset$$

$$\text{SUM}_2 \quad \frac{ge_2 \Downarrow \nu\vec{n}.\alpha.e}{ge_1 \oplus ge_2 \Downarrow \nu\vec{n}.\alpha.e} \quad \{\vec{n}\} \cap \text{fn}(ge_1) = \emptyset$$

$$\text{WRAP} \quad \frac{ge \Downarrow \nu\vec{n}.\alpha.e}{ge \Rightarrow v \Downarrow \nu\vec{n}.\alpha.ve} \quad \{\vec{n}\} \cap \text{fn}(v) = \emptyset$$

Figure 8.8: Evaluation Relation on CML_ν — Guarding Rules

The second group, given in figure 8.8, are those dealing with guarded processes. We do not give the rules for restriction here, but instead collect them together in figure 8.9. The two (SUM) rules indicate that $\oplus$ is choice, and in particular not internal choice. The rule for wrappers (WRAP) applies a function value to an expression. This function value may be the received value from an input, the trivial function from an output, or the result of another wrapped expression.

The third group, given in figure 8.9 on the opposite page, are those dealing with the restriction operator. We include the (STRUC) rule because the structural equivalence deals mainly with the scope of restriction. The other rules deal with the different cases depending on whether the process outputs or returns a value, and whether the name being restricted is used in the value. A name is only allowed in the initial restriction if there is a free variable of the same name in the value that the process outputs or returns.

In figure 8.10 on page 110 we give the static rules. These are the rules where a larger context does not inhibit the reduction of an inner part of the process. It is slightly more restrictive than evaluation contexts for two reasons. The first reason is that a value may only be returned from the rightmost process. The second is that, apart from (PAR), the value will be used by the context. These two reasons mean that we only allow a value in rule (PAR_1).

$$\text{STRUC}\quad \frac{f \equiv e \quad e \Downarrow \nu\vec{n}.l.e' \quad e' \equiv f'}{f \Downarrow \nu\vec{n}'.l.f'}$$

$$\text{RES}_? \quad \frac{e \Downarrow n?x.e'}{\nu m.e \Downarrow n?x.\nu m.e'} \qquad n \neq m$$

$$\text{rRES}_\surd \quad \frac{e \Downarrow \nu\vec{m}.\surd v.e'}{\nu n.e \Downarrow \nu n.\nu\vec{m}.\surd v.e'} \qquad n \in \text{fn}(v) \setminus \vec{m}$$

$$\text{rRES}_! \quad \frac{e \Downarrow \nu\vec{m}.n!v.e'}{\nu m'.e \Downarrow \nu m'.\nu\vec{m}.n!v.e'} \qquad n \neq m', \quad m' \in \text{fn}(v) \setminus \vec{m}$$

$$\text{uRES}_\surd \quad \frac{e \Downarrow \nu\vec{m}.\surd v.e'}{\nu n.e \Downarrow \nu\vec{m}.\surd v.\nu n.e'} \qquad n \notin \text{fn}(v)$$

$$\text{uRES}_! \quad \frac{e \Downarrow \nu\vec{m}.n!v.e'}{\nu m'.e \Downarrow \nu\vec{m}.n!v.\nu m'.e'} \qquad m' \notin \{n, \text{fn}(v)\}$$

Figure 8.9: Evaluation Relation on CML_ν — Restriction Rules

The final set of rules are contained in figure 8.11 on page 111. These are the rules that give the evaluation relation its "big step" nature. They are mainly rules where a result is produced by an inner process and used by its context to determine how the evaluation will proceed. There are two main, related, points to note. Firstly, as in the evaluation relation for NCCS, a subterm may only reduce if the whole term then has an action which it may perform. In general a subterm which may evaluate to a value does not necessarily have a continuation which may further reduce. The second point is that of the evaluation contexts which deal with this apparent problem. As previously mentioned, the evaluation context is such that it does not inhibit reduction of the inner term. Here it is being used to show that if the inner term performs a reduction then the whole term may perform that reduction and then continue, but only if the whole term (after the reduction) has an action that it may perform.

We also require a definition for δ which is used in ($\text{CON}_\surd$). Obviously δ depends on what constant functions there are. We give, in figure 8.12, the

$$\text{PAR}_1 \quad \frac{e \Downarrow \nu\vec{n}.l.e'}{f|e \Downarrow \nu\vec{n}.l.(f|e')} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{PAR}_2 \quad \frac{e \Downarrow \nu\vec{n}.\alpha.e'}{e|f \Downarrow \nu\vec{n}.\alpha.(e'|f)} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{CON}_\alpha \quad \frac{e \Downarrow \nu\vec{n}.\alpha.e'}{ce \Downarrow \nu\vec{n}.\alpha.(ce')}$$

$$\text{APP}_\alpha \quad \frac{e \Downarrow \nu\vec{n}.\alpha.e'}{ef \Downarrow \nu\vec{n}.\alpha.(e'f)} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{PAIR}_\alpha \quad \frac{e \Downarrow \nu\vec{n}.\alpha.e'}{(e,f) \Downarrow \nu\vec{n}.\alpha.(e',f)} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{IF}_\alpha \quad \frac{e \Downarrow \nu\vec{n}.\alpha.e'}{\textbf{if } e \textbf{ then } f \textbf{ else } g \Downarrow \nu\vec{n}.\alpha.(\textbf{if } e' \textbf{ then } f \textbf{ else } g)} \qquad \{\vec{n}\} \cap \text{fn}(f,g) = \emptyset$$

$$\text{LET}_\alpha \quad \frac{e \Downarrow \nu\vec{n}.\alpha.e'}{\textbf{let } x{=}e \textbf{ in } f \Downarrow \nu\vec{n}.\alpha.(\textbf{let } x{=}e' \textbf{ in } f)} \qquad \{\vec{n}\} \cap \text{fn}(f,g) = \emptyset$$

Figure 8.10: Evaluation Relation on CML_ν — Static Rules

evaluation of δ expressions for the constant functions given in figure 8.2 on page 102.

The only difference from [FHJ95] is in dealing with the constant always. This is because we do not have A in our calculus. We could add A into CML_ν, with $\text{A}v$ being a guarded expression, for any value v. The evaluation relation would then need to be augmented with the following rule:

$$ALW \quad \frac{v \Downarrow C}{\text{A}v \Downarrow C}$$

This differs from [FHJ95] in that we combine two steps, the first being "$\text{A}v \to v$" and the second being the evaluation of $v \Downarrow \surd v.\Lambda$. We know that $\text{A}v$ can always evaluate since v can always evaluate to $\surd v.\Lambda$.

Alternatively we may view $\text{A}v$ as being "$\tau.v$". This is essentially how A is seen in [FHJ95]. We may then translate expressions involving A in a similar

$$\text{CON}_{\surd} \quad \frac{e \Downarrow \nu\vec{n}.\surd v.e' \quad \mathcal{E}[\nu\vec{n}.(e'|\delta(c,v))] \Downarrow C}{\mathcal{E}[ce] \Downarrow C}$$

$$\text{IF}_t \quad \frac{e \Downarrow \surd\mathbf{true}.e' \quad \mathcal{E}[e'|e_t] \Downarrow C}{\mathcal{E}[\mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f] \Downarrow C}$$

$$\text{IF}_f \quad \frac{e \Downarrow \surd\mathbf{false}.e' \quad \mathcal{E}[e'|e_f] \Downarrow C}{\mathcal{E}[\mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f] \Downarrow C}$$

$$\text{PAIR}_{\surd} \quad \frac{e_1 \Downarrow \nu\vec{n}.\surd v.e_1' \quad \mathcal{E}[\nu\vec{n}.(e_1'|\mathbf{let}\ x = e_2\ \mathbf{in} < v, x >)] \Downarrow C}{\mathcal{E}[(e_1, e_2)] \Downarrow C} \qquad \{\vec{n}\} \cap \text{fn}(e_2) = \emptyset$$

$$\text{APP}_{\surd} \quad \frac{e_1 \Downarrow \nu\vec{n}.\surd v.e_1' \quad \mathcal{E}[\nu\vec{n}.(e_1'|\mathbf{let}\ y = e_2\ \mathbf{in}\ e_3[v/x])] \Downarrow C}{\mathcal{E}[e_1 e_2] \Downarrow C} \qquad \{\vec{n}\} \cap \text{fn}(e_2, e_3) = \emptyset\ \mathbf{and}\ v = \mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow e_3)$$

$$\text{LET}_{\surd} \quad \frac{e_1 \Downarrow \nu\vec{n}.\surd v.e_1' \quad \mathcal{E}[\nu\vec{n}.(e_1'|e_2[v/x])] \Downarrow C}{\mathcal{E}[\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2] \Downarrow C} \qquad \{\vec{n}\} \cap \text{fn}(e_2) = \emptyset$$

$$\text{COM}_1 \quad \frac{e \Downarrow \nu\vec{m}.n!v.e' \quad f \Downarrow n?x.f' \quad \mathcal{E}[\nu\vec{m}.(e'|f'[v/x])] \Downarrow C}{\mathcal{E}[e|f] \Downarrow C} \qquad \{\vec{m}\} \cap \text{fn}(f) = \emptyset$$

$$\text{COM}_2 \quad \frac{e \Downarrow n?x.e' \quad f \Downarrow \nu\vec{m}.n!v.f' \quad \mathcal{E}[\nu\vec{m}.(e'[v/x]|f')] \Downarrow C}{\mathcal{E}[e|f] \Downarrow C} \qquad \{\vec{m}\} \cap \text{fn}(e) = \emptyset$$

Figure 8.11: Evaluation Relation on CML_ν — Big Step Rules

$$
\begin{array}{rclrcl}
\delta(\mathbf{fst}, < v, w >) & = & v & \delta(\mathbf{snd}, < v, w >) & = & w \\
\delta(\mathbf{add}, < m, n >) & = & m + n & \delta(\mathbf{mul}, < m, n >) & = & m \times n \\
\delta(\mathbf{leq}, < m, n >) & = & m \leq n & & &
\end{array}
$$

$$
\begin{array}{rclrcl}
\delta(\mathbf{transmit}, < k, v >) & = & [k!v] & \delta(\mathbf{receive}, k) & = & [k?] \\
\delta(\mathbf{choose}, < [ge_1], [ge_2] >) & = & [ge_1 \oplus ge_2] & \delta(\mathbf{never}, ()) & = & [\Lambda] \\
\delta(\mathbf{wrap}, < [ge], v >) & = & [ge \Rightarrow v] & \delta(\mathbf{always}, v) & = & v \\
\delta(\mathbf{spawn}, v) & = & v()|() & \delta(\mathbf{sync}, [ge]) & = & ge
\end{array}
$$

$$
\delta(\mathbf{match}, < m, n >) = \begin{array}{ll} \mathbf{true} & \text{if } m = n \\ \mathbf{false} & \text{otherwise} \end{array}
$$

Figure 8.12: Evaluation Relation on CML_ν — Constants

manner to the translation of τNCCS into NCCS (see definition 2.22). There are various possibilities including a generalised form of

$$\mathbf{A}v \stackrel{\triangle}{=} \nu n.(n!v|n?)$$

where n is a fresh name. This may then be extended so that $\mathbf{A}v$ may be part of a sum or a wrapped expression. For example:

$$G \oplus (\mathbf{A}v \Rightarrow v') \stackrel{\triangle}{=} \nu n.(n!v|G \oplus (n? \Rightarrow v')$$

In this particular case one fresh name would be used for each **A**. If we wished to reduce the number of fresh names needed we could instead translate $\mathbf{A}v$ by

$$\mathbf{A}v \stackrel{\triangle}{=} \nu n.(\{n? \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)\}|\{n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow v)\})$$

where x is not a free name or variable in v.

If we use this encoding then it can also be extended, for example:

$$G \oplus (\mathbf{A}v \Rightarrow v') \stackrel{\triangle}{=} \nu n.\,(\{n? \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)\}\,|G \oplus \{n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow v)\})$$

8.5 Example CML_ν program

As an example we will consider the problem of a paranoid executive. He is the manager of a large, hi-tech company. Not only is he fed up with unwanted

e-mail but he is also worried about the possibility of unsolicited telephone calls. A solution that he is considering is that when a customer wants to phone him they are issued with their own private (internal) telephone code. This number will connect them to a relay which is, in turn, connected to the executive's telephone. The reason for the private number is that the executive wants to be able to track the number if it is passed around. An additional requirement is that the executive must be able to disable the relay. This will mean that if someone gives their private number out then the relay can be disabled and they will be unable to connect to the relay in future.

The following is presented as a possible solution, together with some discussion about the problems with it. We regard all of the executive's side of the problem as being dealt with by a "manager expression". This will log all requests for a new private number, together with the information required to disable the relay. We do not concern ourselves with the manager expression but merely look at a function which we will call the Initiator, In. In order to set up the Initiator we assume that there are two specific channel names. These are m and m'. The first is used as a control channel. It is used for communication with the manager expression. The second is the executive's real "telephone" number. Neither of these should be viewed as public, but we do not specifically restrict them. The main reason for this is that we do not want to include the manager expression. This will allow us to focus on the Initiator function.

The expression In is a function which will do two jobs. It will set up the relay, R, and return the private "telephone number" (channel name). It also communicates the details of the relay to the manager expression. The function takes one argument, which is the "telephone number" of the person who wishes to set up the link. This "telephone number" will be used to send messages back from the executive to the customer.

$$In = \mathbf{fn}\,(x \Rightarrow \nu p.\nu d.(m! << p, d >, x >\Rightarrow \mathbf{fn}\,(y \Rightarrow R|p)))$$

Informally the function takes one argument, the customer's "telephone number", and then creates two private channel names, p and d. The first is the private number and the second is used to disable the relay. It then tells the manager expression about both these private channel names, as well as the

customer's "telephone number". After this it starts up the relay and returns the private "telephone number" which the customer should use to telephone the executive.

More formally in figure 8.13 on the opposite page we give a possible evaluation. Noting that R, given below, contains no free variables, we use (in order from top to bottom): (OUT), (WRAP), (rRES$_!$) twice, (STRUC), (LET$_\surd$), (APP$_\surd$) and (VAL) twice. First a nested pair, containing the two new names and the argument to C, is outputted on channel m; shown in step 1. We remember that m should be viewed as a restricted channel, and so the manager expression will need to input the nested pair. The continuation can then evaluate as shown in step 2, using (VAL) three times, (PAR$_1$), (uRES$_\surd$), (rRES$_\surd$), (STRUC), (LET$_\surd$) and (APP$_\surd$). Here the relay R is spawned and the "private telephone number" is returned. We note that the remaining continuation cannot now produce any more values, but this does not mean that it cannot do anything. The relay is still active. This is an example of an expression that can start up a "background process".

There is an obvious problem with this function. This is that, being a function, it can be used many times. This could easily be remedied by making the "manager expression" input on channel name m only once. This would require making a new channel name for each function. However that is not a problem since we have dynamic name creation. We have to create a new name anyway since we want m to be a private name.

We now consider the implementation of the relay R. We divide R into three different parts. The first, D (Disable), allows the relay to be disabled. The other two, L (Listen) and A (Answer) act as the relay.

$$\begin{aligned} R &::= \mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow ((D \oplus L) \oplus A))\,() \\ D &::= d? \Rightarrow \mathbf{fn}\,(z \Rightarrow \Lambda) \\ L &::= p? \Rightarrow \mathbf{fn}\,(z \Rightarrow (m'! < p, z > \Rightarrow \mathbf{fn}\,(w \Rightarrow x))) \\ A &::= m'? \Rightarrow \mathbf{fn}\,(z \Rightarrow (t!z \Rightarrow \mathbf{fn}\,(w \Rightarrow x))) \end{aligned}$$

Again we describe the expression informally first. We will first describe the expressions, D, L and A, starting with D. D will read in a value on channel d and then become the dead process. In this way the "manager" may send a message to the relay on channel d and the channel will then die. L reads

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{m! << p,d >,t > \Downarrow m! << p,d >,t > .()}}{m! << p,d >,t > \Rightarrow \mathbf{fn}\,(y \Rightarrow (R|p))\,() \Downarrow m! << p,d >,t > .\{\mathbf{fn}\,(y \Rightarrow (R|p))\,()\}}}{\nu d.\{m! << p,d >,t > \Rightarrow \mathbf{fn}\,(y \Rightarrow (R|p))\} \Downarrow \nu d.m! << p,d >,t > .\{\mathbf{fn}\,(y \Rightarrow (R|p))\,()\}}}{\nu p.\nu d.\{m! << p,d >,t > \Rightarrow \mathbf{fn}\,(y \Rightarrow (R|p))\} \Downarrow \nu p.\nu d.m! << p,d >,t > .\{\mathbf{fn}\,(y \Rightarrow (R|p))\,()\}}}{\Lambda|\Lambda|\nu p.\nu d.\{m! << p,d >,x > \Rightarrow \mathbf{fn}\,(y \Rightarrow (R|p))\}[t/x] \Downarrow \nu p.\nu d.m! << p,d >,t > .\{\mathbf{fn}\,(y \Rightarrow (R|p))\,()\}}}{\Lambda|\mathbf{let}\ x = t\ \mathbf{in}\ \nu p.\nu d.\{m! << p,d >,x > \Rightarrow \mathbf{fn}\,(y \Rightarrow (R|p))\}[In/z] \Downarrow \nu p.\nu d.m! << p,d >,t > .\{\mathbf{fn}\,(y \Rightarrow (R|p))\,()\}}$$

$$\dfrac{In \Downarrow \surd In.\Lambda \qquad \dfrac{t \Downarrow \surd t.\Lambda \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \Downarrow \surd p.\Lambda}}{R|p \Downarrow \surd p.(R|\Lambda)}}{\nu d.(R|p) \Downarrow \surd p.\nu d.(R|\Lambda)}}{\nu p.\nu d.(R|p) \Downarrow \nu p.\surd p.\nu d.(R|\Lambda)}}{\nu p.\nu d.(\Lambda|\Lambda|(R|p)[()/y]) \Downarrow \nu p.\surd p.\nu d.(R|\Lambda)}}{(In)t \Downarrow \nu p.\nu d.m! << p,d >,t > .\{\mathbf{fn}\,(y \Rightarrow (R|p))\,()\}}$$

Step 1

$$\dfrac{\dfrac{\dfrac{}{\mathbf{fn}\,(y \Rightarrow (R|p)) \Downarrow \surd \mathbf{fn}\,(y \Rightarrow (R|p))\,.\Lambda} \qquad \dfrac{\dfrac{}{() \Downarrow \surd().\Lambda} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \Downarrow \surd p.\Lambda}}{R|p \Downarrow \surd p.(R|\Lambda)}}{\nu d.(R|p) \Downarrow \surd p.\nu d.(R|\Lambda)}}{\nu p.\nu d.(R|p) \Downarrow \nu p.\surd p.\nu d.(R|\Lambda)}}{\nu p.\nu d.(\Lambda|\Lambda|(R|p)[()/y]) \Downarrow \nu p.\surd p.\nu d.(R|\Lambda)}}{\nu p.\nu d.(\Lambda|\mathbf{let}\ y = ()\ \mathbf{in}\ (R|p)) \Downarrow \nu p.\surd p.\nu d.(R|\Lambda)}}{\nu p.\nu d.\mathbf{fn}\,(y \Rightarrow (R|p))\,() \Downarrow \nu p.\surd p.\nu d.(R|\Lambda)}$$

Step 2

Figure 8.13: Example evaluation — Initiator

in a "message" on channel p, sends the "message" to the manager and then becomes R. Similarly A reads in a message from the "manager", sends it out on channel t, and then becomes R. R, then, may do any one of D, L and A. Since D ends up with the dead process and not R, we see that it does indeed disable the relay.

To look at the relay more formally we consider it receiving a value on each of the three channels, d, p and m'. To reduce repetition we let R' be the function such that $R = R'()$. For case D there are two steps; input on channel d and then a clean-up step, both given in figure 8.14 on the next page. The latter does not input or output so we need another process to generate a committed form. The input step uses rules (IN), (WRAP), (SUM_1) twice, (STRUC), ($\text{LET}_\surd$), ($\text{APP}_\surd$) and (VAL) twice. The clean-up step uses (OUT), (PAR_2), (STRUC), ($\text{LET}_\surd$), ($\text{APP}_\surd$) and (VAL) twice. We can see from the derivations that K does indeed do what was claimed. L and A both have a three step sequence of evaluations. First they read in a value on p (m') and then output on m' (t respectively) followed by a clean-up step. When a sequence of values is sent, the clean-up step may be joined onto the front of a read step, however we keep them separate for clarity. Because A and L are essentially the same, we only give the steps for L.

From the derivations given in figure 8.15 on page 118, we see that the listen (and similarly the answer) part of the relay behaves as required. The derivations proceed using: (IN), (WRAP), (SUM_2), (SUM_1), (STRUC), ($\text{LET}_\surd$), ($\text{APP}_\surd$) and (VAL) twice for the input step. Then (OUT), (WRAP), (STRUC), ($\text{LET}_\surd$), ($\text{APP}_\surd$) and (VAL) twice are used for the output step. Finally, (OUT), (PAR_2), (STRUC), ($\text{LET}_\surd$), ($\text{APP}_\surd$) and (VAL) twice are used for the clean-up step. One point to note is that when a message is sent on m', it is paired with the channel name that it was originally received on. This means that the executive can tell who sent the message. At least he can if the customer has not shared his code, in which case he only knows who the code was originally given to.

There are also problems with the relay. One is that m' is not a private name. This could be overcome by restricting the name. Alternatively the relay could be started up by the "manager" expression.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{d?\Downarrow d?w.w}}{K\Downarrow d?w.\mathbf{fn}\,(z\Rightarrow\Lambda)\,w}}{K\oplus L[R'/x]\Downarrow d?w.\mathbf{fn}\,(z\Rightarrow\Lambda)\,w}}{((K\oplus L[R'/x])\oplus A[R'/x])[()/y]\Downarrow d?w.\mathbf{fn}\,(z\Rightarrow\Lambda)\,w}}{\dfrac{\dfrac{}{()\surd().\Lambda}\qquad\Lambda|\Lambda|((K\oplus L[R'/x])\oplus A[R'/x])[()/y]\Downarrow d?w.\mathbf{fn}\,(z\Rightarrow\Lambda)\,w}{\dfrac{\dfrac{}{R'\Downarrow\surd R'.\Lambda}\qquad\Lambda|\mathbf{let}\,y=()\,\mathbf{in}\,((K\oplus L)\oplus A)[R'/x]\Downarrow d?w.\mathbf{fn}\,(z\Rightarrow\Lambda)\,w}{R'()\Downarrow d?w.\mathbf{fn}\,(z\Rightarrow\Lambda)\,w}}}$$

Value sent on d — Input step

$$\dfrac{\dfrac{\dfrac{}{\mathbf{fn}\,(z\Rightarrow\Lambda)\Downarrow\surd\mathbf{fn}\,(z\Rightarrow\Lambda)\,.\Lambda}\qquad\dfrac{\dfrac{}{w\Downarrow\surd w.\Lambda}\qquad\dfrac{\dfrac{\dfrac{}{a!\mathbf{true}\Downarrow a!\mathbf{true}.()}}{a!\mathbf{true}|\Lambda\Downarrow a!\mathbf{true}.(()|\Lambda)}}{a!\mathbf{true}|\Lambda|\Lambda|\Lambda[w/x]\Downarrow a!\mathbf{true}.(()|\Lambda)}}{a!\mathbf{true}|\Lambda|\mathbf{let}\,x=w\,\mathbf{in}\,\Lambda[K/z]\Downarrow a!\mathbf{true}.(()|\Lambda)}}{a!\mathbf{true}|\mathbf{fn}\,(z\Rightarrow\Lambda)\,w\Downarrow a!\mathbf{true}.(()|\Lambda)}$$

Value sent on d — Clean up step

Figure 8.14: Example evaluation — Relay, disable

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p?\Downarrow p?u.u}}{L[R'/x]\Downarrow p?u.(\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))))u}}{K\oplus L[R'/x]\Downarrow p?u.(\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))))u}}{((K\oplus L[R'/x])\oplus A[R'/x])[()/y]\Downarrow p?u.(\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))))u}}{\Lambda|\Lambda|((K\oplus L[R'/x])\oplus A[R'/x])[()/y]\Downarrow p?u.(\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))))u}\quad\dfrac{}{()\surd().\Lambda}}{\Lambda|\mathbf{let}\,y=()\,\mathbf{in}\,((K\oplus L)\oplus A)[R'/x]\Downarrow p?u.(\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))))u}\quad\dfrac{}{R'\Downarrow\surd R'.\Lambda}}{R'()\Downarrow p?u.(\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))))u}$$

Value sent on p — Input step

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{m!<p,u>\Downarrow m!<p.u>.()}}{m!<p,u>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')\Downarrow m!<p.u>.\mathbf{fn}\,(w\Rightarrow R')\,()}}{\Lambda|\Lambda|(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R'))[u/z]\Downarrow m!<p.u>.\mathbf{fn}\,(w\Rightarrow R')\,()}\quad\dfrac{}{u\Downarrow\surd u.\Lambda}}{\Lambda|\mathbf{let}\,z=w\,\mathbf{in}\,(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R'))\Downarrow m!<p.u>.\mathbf{fn}\,(w\Rightarrow R')\,()}\quad\dfrac{}{L'\Downarrow\surd L'.\Lambda}}{\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))\,u\Downarrow m!<p.u>.\mathbf{fn}\,(w\Rightarrow R')\,()}$$

where $L'=\mathbf{fn}\,(z\Rightarrow(m!<p,z>\Rightarrow\mathbf{fn}\,(w\Rightarrow R')))$.

Value sent on p — Output step

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a!\mathbf{true}\Downarrow a!\mathbf{true}.()}}{a!\mathbf{true}|R'\Downarrow a!\mathbf{true}.(()|R')}}{a!\mathbf{true}|\Lambda|\Lambda|R'[()/w]\Downarrow a!\mathbf{true}.(()|R')}\quad\dfrac{}{()\Downarrow\surd().\Lambda}}{a!\mathbf{true}|\Lambda|\mathbf{let}\,w=()\,\mathbf{in}\,R'[\mathrm{fn}(w)R'/z]\Downarrow a!\mathbf{true}.(()|R')}\quad\dfrac{}{\mathbf{fn}\,(w\Rightarrow R')\Downarrow\surd\mathbf{fn}\,(w\Rightarrow R').\Lambda}}{a!\mathbf{true}|\mathbf{fn}\,(w\Rightarrow R')\,()\Downarrow a!\mathbf{true}.(()|R')}$$

Value sent on p — Clean up step

Figure 8.15: Example evaluation — Relay, listen

Chapter 9

Transition relation for CML$_\nu$

Definition 9.1

In the same manner as in definition 8.7, we let $\alpha_\tau = n!v|n?x|\tau$, $l = \alpha_\tau|\surd v$ and $C = \nu\vec{n}.l.e$ with $\{\vec{n}\} = \emptyset$ if $l = n?x$ or $l = \tau$.

We now define the "small step" transition relation for CML$_\nu$. Again, we define it in terms of the same five groups as for the evaluation relation.

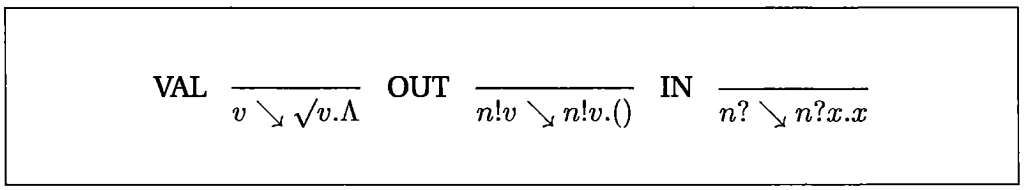

Figure 9.1: Transition Relation on CML$_\nu$ — Axioms

The rules for axioms are the same as those used in the evaluation relation. This is because we are using the same syntax for both the evaluation relation and the transition relation.

Similarly the rules for guarded expressions are essentially the same as for the evaluation relation, as are the rules for restriction and the static rules.

The "small step" nature of the transition relation follows from the silent rules. Instead of continuing the reduction in the presence of an evaluation context, the reduction stops. We observe (in lemma 9.30 on page 136) that not having the evaluation context in the silent rules does not reduce the possible transitions.

We use the same evaluation relation for δ as given in figure 8.12.

$$\text{SUM}_1 \quad \frac{ge_1 \searrow \nu\vec{n}.\alpha.e}{ge_1 \oplus ge_2 \searrow \nu\vec{n}.\alpha.e} \quad \{\vec{n}\} \cap \text{fn}(ge_2) = \emptyset$$

$$\text{SUM}_2 \quad \frac{ge_2 \searrow \nu\vec{n}.\alpha.e}{ge_1 \oplus ge_2 \searrow \nu\vec{n}.\alpha.e} \quad \{\vec{n}\} \cap \text{fn}(ge_1) = \emptyset$$

$$\text{WRAP} \quad \frac{ge \searrow \nu\vec{n}.\alpha.e}{ge \Rightarrow v \searrow \nu\vec{n}.\alpha.ve} \quad \{\vec{n}\} \cap \text{fn}(v) = \emptyset$$

Figure 9.2: Transition Relation on CML$_\nu$ — Guarding Rules

$$\text{STRUC} \quad \frac{f \equiv e \quad e \searrow \nu\vec{n}.l.e' \quad e' \equiv f'}{f \searrow \nu\vec{n}.l.f'}$$

$$\text{RES}_? \quad \frac{e \searrow n?x.e'}{\nu m.e \searrow n?x.\nu m.e'} \quad n \neq m$$

$$\text{rRES}_\surd \quad \frac{e \searrow \nu\vec{m}.\surd v.e'}{\nu n.e \searrow \nu n.\nu\vec{m}.\surd v.e'} \quad n \in \text{fn}(v)$$

$$\text{rRES}_! \quad \frac{e \searrow \nu\vec{m}.n!v.e'}{\nu m'.e \searrow \nu m'.\nu\vec{m}.n!v.e'} \quad n \neq m', \quad m' \in \text{fn}(v)$$

$$\text{uRES}_\surd \quad \frac{e \searrow \nu\vec{m}.\surd v.e'}{\nu n.e \searrow \nu\vec{m}.\surd v.\nu n.e'} \quad n \notin \text{fn}(v)$$

$$\text{uRES}_! \quad \frac{e \searrow \nu\vec{m}.n!v.e'}{\nu m'.e \searrow \nu\vec{m}.n!v.\nu m'.e'} \quad m' \notin \{n, \text{fn}(v)\}$$

$$\text{RES}_\tau \quad \frac{e \searrow \tau.e'}{\nu n.e \searrow \tau.\nu n.e'}$$

Figure 9.3: Transition Relation on CML$_\nu$ — Restriction Rules

$$\text{PAR}_1 \quad \frac{e \searrow \nu\vec{n}.l.e'}{f|e \searrow \nu\vec{n}.l.(f|e')} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{PAR}_2 \quad \frac{e \searrow \nu\vec{n}.\alpha_\tau.e'}{e|f \searrow \nu\vec{n}.\alpha_\tau.(e'|f)} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{CON}_\alpha \quad \frac{e \searrow \nu\vec{n}.\alpha_\tau.e'}{ce \searrow \nu\vec{n}.\alpha_\tau.(ce')} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{APP}_\alpha \quad \frac{e \searrow \nu\vec{n}.\alpha_\tau.e'}{ef \searrow \nu\vec{n}.\alpha_\tau.(e'f)} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{PAIR}_\alpha \quad \frac{e \searrow \nu\vec{n}.\alpha_\tau.e'}{(e,f) \searrow \nu\vec{n}.\alpha_\tau.(e',f)} \qquad \{\vec{n}\} \cap \text{fn}(f) = \emptyset$$

$$\text{IF}_\alpha \quad \frac{e \searrow \nu\vec{n}.\alpha_\tau.e'}{\textbf{if } e \textbf{ then } f \textbf{ else } g \searrow \nu\vec{n}.\alpha_\tau.(\textbf{if } e' \textbf{ then } f \textbf{ else } g)} \qquad \{\vec{n}\} \cap \text{fn}(f,g) = \emptyset$$

$$\text{LET}_\alpha \quad \frac{e \searrow \nu\vec{n}.\alpha_\tau.e'}{\textbf{let } x = e \textbf{ in } f \searrow \nu\vec{n}.\alpha_\tau.(\textbf{let } x = e' \textbf{ in } f)} \qquad \{\vec{n}\} \cap \text{fn}(f,g) = \emptyset$$

Figure 9.4: Transition Relation on CML_ν — Static Rules

$$\text{CON}_\surd \quad \frac{e \searrow \nu\vec{n}.\surd v.e'}{ce \searrow \tau.\nu\vec{n}.(e'|\delta(c,v))}$$

$$\text{IF}_t \quad \frac{e \searrow \surd\mathbf{true}.e'}{\mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \searrow \tau.(e'|e_t)}$$

$$\text{IF}_f \quad \frac{e \searrow \surd\mathbf{false}.e'}{\mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \searrow \tau.(e'|e_f)}$$

$$\text{PAIR}_\surd \quad \frac{e_1 \searrow \nu\vec{n}.\surd v.e_1'}{(e_1,e_2) \searrow \tau.\nu\vec{n}.(e'|\mathbf{let}\ x=e_2\ \mathbf{in} < v,x >)} \quad \{\vec{n}\} \cap \text{fn}(e_2) = \emptyset$$

$$\text{APP}_\surd \quad \frac{e_1 \searrow \nu\vec{n}.\surd v.e_1'}{e_1e_2 \searrow \tau.\nu\vec{n}.(e_1'|\mathbf{let}\ y=e_2\ \mathbf{in}\ e_3[v/x])} \quad \{\vec{n}\} \cap \text{fn}(e_2,e_3) = \emptyset \text{ and } v = \mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow e_3)$$

$$\text{LET}_\surd \quad \frac{e_1 \searrow \nu\vec{n}.\surd v.e_1'}{\mathbf{let}\ x=e_1\ \mathbf{in}\ e_2 \searrow \tau.\nu\vec{n}.(e_1'|e_2[v/x])} \quad \{\vec{n}\} \cap \text{fn}(e_2) = \emptyset$$

$$\text{COM}_1 \quad \frac{e \searrow \nu\vec{m}.n!v.e' \quad f \searrow n?x.f'}{e|f \searrow \tau.\nu\vec{m}.(e'|f'[v/x])} \quad \{\vec{m}\} \cap \text{fn}(f) = \emptyset$$

$$\text{COM}_2 \quad \frac{e \searrow n?x.e' \quad f \searrow \nu\vec{m}.n!v.f'}{e|f \searrow \tau.\nu\vec{m}.(e'[v/x]|f')} \quad \{\vec{m}\} \cap \text{fn}(e) = \emptyset$$

Figure 9.5: Transition Relation on CML$_\nu$ — Silent Rules

Definition 9.2

We write $e \searrow \tau^*.e'$ to mean that there are e_i such that $e = e_0$, $e' = e_n$ and $e_i \searrow \tau.e_{i+1}$. Similarly we write $e \searrow \tau^*\nu\vec{n}.l.e'$ to mean that $e \searrow \tau^*.e''$ and $e'' \searrow \nu\vec{n}.l.e'$.

Definition 9.3

A derivation of $e \searrow \nu\vec{n}.l.e'$ is *normal* if the (STRUC) rule is used only once, and then as the last rule of the derivation. Furthermore, it is *simple* if it does not use the (STRUC) rule at all. As in definition 4.2 we say that $e \searrow \nu\vec{n}.l.e'$ is normal/simple if the derivation we will use is normal/simple.

We will now define the restriction contexts which will be used in various proofs. These are essentially those contexts which restrictions can be pulled out of or pushed into.

Definition 9.4

The *restriction contexts*, $\mathcal{R}[-]$ are given in figure 9.6.

$$
\begin{array}{lcll}
\mathcal{R}[-] & ::= & - & \\
& | & \mathcal{G}[-] & \\
& | & \nu n.\mathcal{R}[-] & \\
& | & \mathcal{R}[-]|e & \\
& | & e|\mathcal{R}[-] & \\
& | & c\mathcal{R}[-] & \\
& | & \mathcal{R}[-]e & \\
& | & e\mathcal{R}[-] & \\
& | & (\mathcal{R}[-], e) & \\
& | & (e, \mathcal{R}[-]) & \\
& | & \textbf{if } \mathcal{R}[-] \textbf{ then } e_t \textbf{ else } e_f & \\
& | & \textbf{if } e \textbf{ then } \mathcal{R}[-] \textbf{ else } e_f & \\
& | & \textbf{if } e \textbf{ then } e_t \textbf{ else } \mathcal{R}[-] & \\
& | & \textbf{let } x = \mathcal{R}[-] \textbf{ in } e & \\
& | & \textbf{let } x = e \textbf{ in } \mathcal{R}[-] & \\
 & & & \\
\mathcal{G}[-] & ::= & - & - \text{ is a guarded expression} \\
& | & g \oplus \mathcal{G}[-] & \\
& | & \mathcal{G}[-] \oplus g & \\
& | & \mathcal{G}[-] \Rightarrow v &
\end{array}
$$

Figure 9.6: Restriction Contexts for CML_ν

Lemma 9.5

For any CML_ν expression e in which all bound names are distinct and distinct from all free names then $e = \mathcal{R}[\nu n.e']$ implies that $e \equiv \nu n.\mathcal{R}[e']$. Furthermore, if $e \searrow \nu \vec{m}.\alpha_\tau.e''$ has a derivation that does not use α-conversion then either $n \in \{\vec{m}\}$ or $e'' \equiv \nu n.e'''$ for some e'''.

Proof

The first part follows by noting that we may pull out a restriction from any restriction context, remembering that all bound names are distinct and distinct from all free names. The second part follows from a simple induction on the depth of the derivation tree. □

Lemma 9.6

For any CML$_\nu$ expressions $\nu n.e$ and e', names $\vec{m}$ and label α_τ then if e has all bound names distinct and distinct from all free names, and there is a derivation for $\nu n.e \searrow \nu\vec{m}.\alpha_\tau.e'$ which doesn't use any α-conversion STRUC rules, then there is a derivation of $\nu n.e \searrow \nu\vec{m}.\alpha_\tau.e''$ that doesn't use any α-conversion STRUC rules nor any STRUC rules that move the restriction of the name n, where e'' is e' if $n \in \{\vec{m}\}$ and $\nu n.e'' \equiv e'$ otherwise. Also if STRUC rules are ignored then the last rule used in the derivation is the RES rule that introduces the restriction of the name n.

Proof

This follows easily from lemma 9.5. □

Notation 9.7

For labels l and l' we write $l \equiv l'$ to mean

- $l = \tau$ if and only if $l' = \tau$.
- $l = n?x$ if and only if $l' = n?x$.
- $l = n!v$ if and only if $l' = n!v'$ and $v \equiv v'$.
- $l = \surd v$ if and only if $l' = \surd v'$ and $v \equiv v'$.

We now present the main lemma of this section. It will be used many times in the rest of the section and also in chapter 12.

Lemma 9.8

If $e \searrow \nu\vec{n}.l.e'$ then there are f and f' with $f \equiv_\alpha e$, $f' \equiv e'$, $l \equiv l'$ and $f \searrow \nu\vec{n}.l'.f'$ simple, where e, e', f and f' are CML$_\nu$ processes. Furthermore, if all bound names are distinct and different from all free names then we may take $f = e$.

Proof

We proceed by a 6 step method. We give an outline of each step:

1. We α-convert to ensure that all bound names are distinct and distinct from all free names.

2. We pull out restrictions.

3. We replace STRUC rules that introduce Λ's with PAR_1 rules (or change the expression involved in the PAR_1 rule already used).

4. We remove the need for STRUC rules that rebracket expressions by reordering PAR and COM rules.

5. We now replace the STRUC rules that delete Λ's by not introducing the Λ in the first place.

6. Finally we push RES rules into their correct position so that we don't need any STRUC rules that deal with the repositioning of channel name restrictions.

The result is that the only part of STRUC that is needed for the left-hand side of the expression is α-conversion. We now give the details.

1. We first α-convert e so that all bound names are different from all free names and all other bound names. This may be done syntactically by ensuring that no restriction operator uses the same name as any other or any free name. In doing this α-conversion we ensure that we do not change any bound names which are in $\vec{n}$[1]. We call this new expression f. We split each STRUC rule into two; one which deals with α-conversion and one which deals with the other parts of structural congruence. We split the α-conversion part into two, the first is the parts that deal with the left-hand part of the transition and the second deals with the right-hand part. We push this first part up the derivation tree[2], adding a STRUC rule each time for the right-hand part of the transition. This follows easily by induction on the depth of the derivation above the last use STRUC rule dealing with α-conversion of

[1]We may do this since no name in $\vec{n}$ may occur free in e, and they must all be distinct.

[2]N.B. the requirement that all bound names are distinct and distinct from all free names is used heavily here since otherwise we might "α-convert" expressions such as $\nu n.(a!\mathbf{true}|n!\mathbf{true})$ into $\nu a.(a!\mathbf{true}|a!\mathbf{true})$ which is not allowed.

the left-hand part of the transition. We then push all the α-conversion uses of the STRUC rule down through the derivation. Again this follows easily by induction on the distance to the bottom of the derivation.

2. We now pull out all restrictions (apart from those enclosed in **fix** expressions). This may be done by lemma 9.6. We are left with a derivation that doesn't use α-conversion apart from the last rule, nor does it use STRUC rules to move restrictions, again apart from the last rule. The last rules in the derivation, ignoring STRUC rules, are now all RES rules.

3. We now consider those parts of the STRUC rules that deal with the introduction of Λ's and will turn each one into a PAR$_1$ rule (or are absorbed into an already present PAR$_1$ rule). We split each STRUC rule into two. The first part deals with the introduction of Λ's, and the second part is for the rest of the STRUC rule. We then split each STRUC that introduces a Λ into one rule for each Λ that is introduced. We now push each rule up to the bottom of the block that it affects. If the structural congruence rule is $e \equiv \Lambda|e$ we split e into e_i's where no e_i contains a parallel composition at the outer level and $e = e_1|\ldots|e_n$. Then we observe where, in this block e_1 is introduced. If it is introduced in a PAR rule then we replace e_1 with $\Lambda|e_1$. Otherwise e_1 must occur on it's own, at the top of the block. We then add an extra PAR$_1$ rule which merely adds Λ. Whichever way the Λ is added we also add a STRUC rule to return the right-hand side of the transition to it's original form, and then push this STRUC rule down the derivation tree. The first part follows by noting that it may all be done syntactically. We may freely push down STRUC rules that do not deal with α-conversion since all bound names are distinct from each other and from all the free names.

4. We now consider each block that is made up of PAR and COM rules. These are delimited by all other rules apart from the STRUC and RES rules. These blocks allow bracketing to be rearranged, either by use of STRUC rules or by the order in which the PAR and COM rules occur.

   We split each STRUC rule up into several rules, each of which deals

with rebracketing of one block, plus one extra rule that deals with all the other uses of the structural congruence. We push the STRUC rule dealing with each block to the end of the set of PAR and COM rules that it deals with. We then split each rule up into many rules, each of which uses only one rule given in figure 8.5 on page 105. We then push each rule up the block as far as it can go. We have three cases to consider.

- The first is that the rebracketing occurs in an expression that is introduced by a PAR rule, in which case we merely rebracket the expression as it is introduced at the cost of a STRUC rule that rebrackets that right-hand side of the transition. This new STRUC rule is then pushed down the derivation.
- The second is that the last two rules before the STRUC were either both PAR rules or one was a PAR rule and the other was a COM rule. In each case we may rearrange the derivation as follows:

$$\dfrac{\dfrac{\dfrac{e \searrow \alpha_\tau.e'}{e|f \searrow \alpha_\tau.(e'|f)}}{(e|f)|g \searrow \alpha_\tau.((e'|f)|g)}}{} \longrightarrow \dfrac{e \searrow \alpha_\tau.e'}{e|(f|g) \searrow \alpha_\tau.(e'|(f|g))}$$

$$\dfrac{\dfrac{\dfrac{f \searrow \alpha_\tau.f'}{e|f \searrow \alpha_\tau.(e|f')}}{(e|f)|g \searrow \alpha_\tau.((e|f')|g)}}{} \longrightarrow \dfrac{\dfrac{f \searrow \alpha_\tau.f'}{f|g \searrow \alpha_\tau.(f'|g)}}{e|(f|g) \searrow \alpha_\tau.(e|(f'|g))}$$

$$\dfrac{\dfrac{\dfrac{e \searrow \alpha.e'}{e|f \searrow \alpha.(e'|f)} \quad g \searrow \overline{\alpha}.g'}{(e|f)|g \searrow \tau.((e'|f)|g')}}{} \longrightarrow \dfrac{e \searrow \alpha.e' \quad \dfrac{g \searrow \overline{\alpha}.g'}{f|g \searrow \overline{\alpha}.(f|g')}}{e|(f|g) \searrow \tau.(e'|(f|g'))}$$

$$\dfrac{\dfrac{\dfrac{f \searrow \alpha.f'}{e|f \searrow \alpha.(e|f')} \quad g \searrow \overline{\alpha}.g'}{(e|f)|g \searrow \tau.((e|f')|g')}}{} \longrightarrow \dfrac{\dfrac{f \searrow \alpha.f' \quad g \searrow \overline{\alpha}.g'}{f|g \searrow \tau.(f'|g')}}{e|(f|g) \searrow \tau.(e|(f'|g'))}$$

$$\dfrac{\dfrac{\dfrac{e \searrow \alpha.e' \quad f \searrow \overline{\alpha}.f'}{e|f \searrow \tau.(e'|f')}}{(e|f)|g \searrow \tau.((e'|f')|g)}}{} \longrightarrow \dfrac{e \searrow \alpha.e' \quad \dfrac{f \searrow \overline{\alpha}.f'}{f|g \searrow \overline{\alpha}.(f'|g)}}{e|(f|g) \searrow \tau.(e'|(f'|g))}$$

The other cases occur symmetrically. Again we have to introduce a

STRUC rule to rebracket the right-hand side of the transition. This STRUC rule is again pushed down the derivation tree.

- The third case occurs when we have $(e|(f|g))$ and one PAR rule introduced both f and g at once. In this case we perform the following transformation:

$$\frac{g \searrow \alpha_\tau.g'}{(e|f)|g \searrow \alpha_\tau.((e|f)|g')} \longrightarrow \frac{\dfrac{g \searrow \alpha_\tau.g'}{f|g \searrow \alpha_\tau.(f|g')}}{e|(f|g) \searrow \alpha_\tau.(e|(f|g'))}$$

Again the symmetric case occurs similarly and we need an extra STRUC rule to rebracket the right-hand side of the transition. This STRUC rule is again pushed down through the derivation tree.

5. We now consider the STRUC rules, that deal with the removal of Λ's, splitting where necessary. We push these rules that remove Λ's up the derivation tree until they reach the PAR rule that introduces them. The rule is then either removed if it only introduces the Λ or the relevant Λ is removed from the expression introduced by the PAR rule.

6. By considering the bracketing of the final expression on the left-hand side of the transition and the placement of the restriction operators, we move the restriction rules up the derivation tree until they are in the correct position. There won't be any problems with bound names becoming free because of step 1. We note that we can put the rule in the exact position required because of the bracketing, although this may involve merging a RES rule with a PAR rule if the restriction only appears in the expression introduced by the PAR rule. As in the previous step we add a STRUC rule to move the restriction operators to their correct places on the right-hand side of the transition. This STRUC rule is again pushed down through the derivation.

This gives us the required simple derivation tree and f'. The second part of the lemma follows by observing that we do not need to α-convert e in order to ensure that all bound names are distinct and different from all free names.

□

Lemma 9.9

Given any normal derivation of $e \searrow \nu\vec{n}.l.e'$, for CML$_\nu$ processes e and e', names $\vec{n}$ and label l, then for any label l' with $l \equiv l'$ there is a normal derivation of $e \searrow \nu\vec{n}.l'.e'$.

Proof

If l is an input or a τ action then the result follows trivially. Otherwise l is $n!v$ or $\surd v$ and l' is $n!v'$ or $\surd v'$ respectively. The result follows by replacing v by v' in the first (topmost) rule used in the derivation and adding an extra STRUC rule at the end, noting that two STRUC rules may be combined into one. □

Lemma 9.10

If $e \searrow \nu\vec{n}.l.e'$, where e and e' are CML$_\nu$ processes, then there is a normal derivation for it.

Proof

The result follows trivially from lemmas 9.8 and 9.9. □

9.1 Initial Lemmas

We prove various lemmas that we will use later. Each lemma follows the same method so we give the general method here and only the particular details of each lemma in its proof. In each case the assumptions and conclusions are only given up to the structural congruence shown in figure 8.5 on page 105, and each lemma involves a single step transition. We only prove these lemmas for CML$_\nu$ processes. We first prove the lemma assuming that the transition is simple. This means that the transition is structure preserving, and we will use this property many times. We may then extend the lemma to the case where the transition is normal, since the assumptions and conclusions involve the structural congruence. We use lemma 9.10 to show that this is enough to prove the lemma for the general case.

Note 9.11

We will use lemma 9.9 many times in this section without explicit reference.

Lemma 9.12

For any CML$_\nu$ expressions e, f and e' and fresh name m then $m!\mathbf{true}|f \equiv e$ and $e \searrow \tau.e'$ imply that there is an f' with $e' \equiv m!\mathbf{true}|f'$ and $f \searrow \tau.f'$.

Proof

We let $g \equiv_\alpha f$ with the additional restrictions that all bound names in g are distinct and different from the free names of g and the fresh name m. Then, using (STRUC), by lemma 9.8 on page 124 there is a simple derivation $m!\mathbf{true}|g \searrow \tau.g''$ with $g'' \equiv e'$. Then, noting that the name m occurs only once in $m!\mathbf{true}|g$, we observe that $m!\mathbf{true}$ must have been introduced by the last PAR$_1$ rule used (taking the whole of the left-hand branch of a COM rule to be later than the right-hand branch). Thus, by deleting this rule if it only introduces $m!\mathbf{true}$ or deleting the $m!\mathbf{true}$ from the rule if not, we obtain a derivation of $g \searrow \tau.g'$ where $g'' = m!\mathbf{true}|g'$. We let $f' = g'$ and then one use of STRUC shows that $f \searrow \tau.f'$, as required. □

Lemma 9.13

For any CML$_\nu$ expressions e, f and e' and fresh name m then $m!\mathbf{true}|f \equiv e$ and $e \searrow m!\mathbf{true}.e'$ imply that $e' \equiv ()|f$.

Proof

We let $g \equiv_\alpha f$ with the additional restrictions that all bound names in g are distinct and different from the free names of g and the fresh name m. Then, using (STRUC), by lemma 9.8 on page 124 there is a simple derivation $m!\mathbf{true}|g \searrow m!\mathbf{true}.g''$ with $g'' \equiv e'$. Then, noting that the name m occurs only once in $m!\mathbf{true}|g$, we observe that the first rule used must have been OUT, and that all the other rules used were PAR$_2$. Therefore we see that $g'' = ()|g$ and the result follows by chasing structural equivalences. □

Corollary 9.14

For any CML$_\nu$ expressions e, f and e' and fresh name m then $m!\mathbf{true}|f \equiv e$ and $e \searrow \tau^*.m!\mathbf{true}.e'$ imply that there is an f' with $e' \equiv ()|f'$ and $f \searrow \tau^*.f'$.

Proof

This follows directly using lemma 9.12 multiple times and lemma 9.13 once. □

We now show that, given any transition, we may remove the outermost restriction operators.

Lemma 9.15

For any CML expressions $\nu\vec{n}.e$ and e', label l and names $\vec{m}$ then $\nu\vec{n}.e \searrow \nu\vec{m}.l.e'$ implies that there is an e'' s.t. $e \searrow \nu\vec{m}'.l.e''$ and $e' \equiv \nu\vec{m}''.e''$, where $\vec{m}' = \vec{m} \setminus \vec{n}$ and $\vec{m}'' = \vec{n} \setminus \vec{m}$.

Proof

By lemma 9.8 there are $f \equiv_\alpha e$ and f' with $e \equiv_\alpha f$, $e' \equiv f'$, $l \equiv l'$ and $f \searrow \nu\vec{m}.l'.f'$ having a simple derivation[3]. We now consider this derivation. The last rules used must have been the RES rules, which may have been of various different types, that introduced $\nu\vec{n}$. Removing these rules gives us a (simple) derivation of $g \searrow \nu\vec{m}'.l'.g'$, where $g \equiv_\alpha e$ and $\nu\vec{m}''.g' \equiv e'$. Therefore taking $e'' = g'$ and using STRUC once, and lemma 9.9, we derive a, normal, derivation of $e \searrow \nu\vec{m}'.l.e''$, as required. □

We now turn our attention to considering what further details we may observe about transitions of parallel compositions of processes. We first show that if a transition is not silent then only one part of the parallel composition is changed[4]. We then go on to describe the various cases that can occur when the transition is silent.

Lemma 9.16

For any expressions e, f and g, label $l \neq \tau$ and names $\vec{n}$ then $e|f \searrow \nu\vec{n}.l.g$ implies that there is a g' with either $e \searrow \nu\vec{n}.l.g'$, $(\nu\vec{n}.g')|f \equiv \nu\vec{n}.g$ and $l \neq \sqrt{v}$ for any v or $f \searrow \nu\vec{n}.l.g'$ and $e|\nu\vec{n}.(g') \equiv \nu\vec{n}.g$.

Proof

By lemma 9.8 there is a simple derivation of $h \searrow \nu\vec{n}.l'.h'$ for some $h \equiv_\alpha e|f$, $l \equiv l'$ and $h' \equiv g$. But we may see that h must be $e'|f'$ for some e' and f', with $\mathrm{bn}(e') \cap \mathrm{bn}(f') = \emptyset$, $e \equiv_\alpha f'$ and $f \equiv_\alpha f'$. The last rule used in the derivation must have been either PAR_1 or PAR_2. The two cases now follow easily, using the fact that $\mathrm{bn}(e') \cap \mathrm{bn}(f') = \emptyset$, lemma 9.9 and STRUC. □

[3]Notice that we may ensure that no name in $\vec{n}$ is α-converted since they can't be the same as any free name.

[4]We define two processes that are structurally congruent to be the same in this context.

Lemma 9.17

For any expressions e, f and g then $e|f \searrow \tau.g$ implies that one of the following four cases must hold:

1. There is an expression f' with $f \searrow \tau.f'$ and $e|f' \equiv g$.
2. There is an expression e' with $e \searrow \tau.e'$ and $e'|f \equiv g$.
3. There are expressions e' and f', value v and names $\vec{m} \subset \mathrm{fn}(v)$ and n with $e \searrow \nu\vec{m}.n!v.e'$, $f \searrow n?x.f'$ and $\nu\vec{m}.(e'|f'[v/x]) \equiv g$.
4. There are expressions e' and f', value v and names $\vec{m} \subset \mathrm{fn}(v)$ and n with $e \searrow n?x.e'$, $f \searrow \nu\vec{m}.n!v.f'$ and $\nu\vec{m}.(e'[v/x]|f') \equiv g$.

Proof

By lemma 9.8 there is a simple derivation of $h \searrow \tau.h'$ for some $h \equiv_\alpha e|f$ and $h' \equiv g$. But we may see that h must be $e''|f''$ for some e'' and f'', with $\mathrm{bn}(e'') \cap \mathrm{bn}(f'') = \emptyset$, $e \equiv_\alpha e''$ and $f \equiv_\alpha f''$. The last rule used in the derivation must have been one of PAR$_1$, PAR $_2$, COM$_1$ and COM$_2$. The four possible rules correspond to the four possible cases, and in each case the result follows immediately. □

We concentrate on how values behave in various parallel contexts. We start by demonstrating that values which are not the rightmost process in an expression behave essentially as if they were "inert", only being subject to α-conversion. We will later show in lemma 10.27 that we may regard the value as not even being there.

Lemma 9.18

For any CML$_\nu$ expressions e and f, value v, label l and names $\vec{n}$ then $v|e \searrow \nu\vec{n}.l.f$ implies that there is an f' with $e \searrow \nu\vec{n}.l.f'$ and $f \equiv v|f'$.

Proof

As in lemma 9.16 overleaf, using lemma 9.8, there are v', l' and e' with $v' \equiv_\alpha v$, $l \equiv l'$ and $e' \equiv_\alpha e$ and $v'|e' \searrow h$ for some $h \equiv \nu\vec{n}.l'.f$. Then the last rule used must have been PAR$_1$. If this rule only introduced v' then the result follows easily. Otherwise we remove the v' from the left-hand side of the introduced expression and the result then follows easily. In each case we use the STRUC rule at most once to α-convert v' back to v. The result then

follows using lemma 9.9. □

Corollary 9.19

For any CML$_\nu$ expressions e, f and e', value v, names $\vec{m}$ and $\vec{n}$ and label l then $e \equiv \nu\vec{m}.(v|f)$ and $e \searrow \nu\vec{n}.l.e'$ imply that there are f', $\vec{m}'$ and $\vec{m}''$ with $\nu\vec{n}.e' \equiv \nu\vec{m}'.\nu\vec{m}''.(v'|f')$, $\nu\vec{m}.(f|v) \searrow \nu\vec{m}'.l.\nu\vec{m}''.(f'|v)$ and $\nu\vec{m}.f \searrow \nu\vec{m}'.l.\nu\vec{m}''.f'$.

Proof

This follows directly from lemmas 9.15 and 9.18. □

Corollary 9.20

For any CML$_\nu$ expressions e, f, g and e', value v, label l and names $\vec{m}$ and $\vec{n}$ then $e \equiv \nu\vec{m}.(f|v|g)$ and $e \searrow \nu\vec{n}.l.e'$ imply that there are f', g', $\vec{m}'$ and $\vec{m}''$ with $\nu\vec{n}.e' \equiv \nu\vec{m}'.\nu\vec{m}''.(f'|v|g')$, $\nu\vec{m}.(f|v|g) \searrow \nu\vec{m}'.l.\nu\vec{m}''.(f'|v|g')$ and $\nu\vec{m}.(f|g) \searrow \nu\vec{m}'.l.\nu\vec{m}''.(f'|g')$.

Proof

First we observe that $e \equiv \nu\vec{m}.(f|(v|g))$ and hence that one use of STRUC implies that $\nu\vec{m}.(f|(v|g)) \searrow \nu\vec{n}.l.e'$. The result then follows from lemmas 9.15, 9.16 and 9.17 and corollary 9.19. □

Corollary 9.21

For any CML$_\nu$ expressions e, f, g and e', label l, names $\vec{n}$ and $\vec{m}$ and value v then

1. $e \equiv \nu\vec{m}.(v|f)$ and $e \searrow \tau^*.\nu\vec{n}.l.e'$ imply that there are f', $\vec{m}'$ and $\vec{m}''$ with $\nu\vec{n}.e' \equiv \nu\vec{m}'.\nu\vec{m}''.(v|f')$, $\nu\vec{m}.(v|f) \searrow \tau^*.\nu\vec{m}'.l.\nu\vec{m}''.(v|f')$ and $\nu\vec{m}.f \searrow \tau^*.\nu\vec{m}'.l.\nu\vec{m}''.f'$.

2. $e \equiv \nu\vec{m}.(f|v|g)$ and $e \searrow \tau^*.\nu\vec{n}.l.e'$ imply there are f', g', $\vec{m}'$ and $\vec{m}''$ with $\nu\vec{n}.e' \equiv \nu\vec{m}'.\nu\vec{m}''.(f'|v|g')$, $\nu\vec{m}.(f|v|g) \searrow \tau^*.\nu\vec{m}'.l.\nu\vec{m}''.(f'|v|g')$ and $\nu\vec{m}.(f|g) \searrow \tau^*.\nu\vec{m}'.l.\nu\vec{m}''.(f'|g')$.

Proof

These follow directly from corollaries 9.19 and 9.20. □

We have now considered the case when the value is not the rightmost process and observed that it acts in an "inert" manner. We may also wonder

how processes behave when they have a value as the rightmost element of the expression. We show that for a process $e|v$ then either e may reduce or the result may reduce to the value v.

Lemma 9.22

For any expressions e, f and e', value v and names $\vec{n}$ and $\vec{m}$ then $e \equiv \nu\vec{m}.(f|v)$ and $e \searrow \nu\vec{n}.\alpha_\tau.e'$ imply that there are f', $\vec{m}'$ and $\vec{m}''$ with $\nu\vec{n}.e' \equiv \nu\vec{m}'.\nu\vec{m}''.(f'|v)$, $\nu\vec{m}.(f|v) \searrow \nu\vec{m}'.\alpha_\tau.\nu\vec{m}''.(f'|v)$ and $\nu\vec{m}.f \searrow \nu\vec{m}'.\alpha_\tau.\nu\vec{m}''.f'$.

Proof

This follows in the same way as Corollary 9.19. □

Lemma 9.23

For any expressions e, f and e', values v and v' and names $\vec{n}$ and $\vec{m}$ then $e \equiv \nu\vec{n}.(f|v)$ and $e \searrow \nu\vec{m}.\surd v'.e'$ imply that $\vec{m} \subseteq \vec{n}$, $v \equiv v'$ and $e' \equiv \nu\vec{m}'.(f|\Lambda)$, where $\vec{m}' = \vec{n} \setminus \vec{m}$.

Proof

This follows from lemma 9.8 noting that the simple derivation must be one use of VAL, then one use of PAR$_1$ and then various RES rules. □

Having shown how values behave in an "inert" way when in the middle of an expression, we now show that we may introduce a value, or more generally any expression, in the middle of a parallel composition. We also show that we may reorder processes under certain conditions.

Lemma 9.24

For CML$_\nu$ expressions e, f, g and e', label l and names $\vec{n}$ and $\vec{m}$ then $e \equiv \nu\vec{m}.(f|g)$ and $e \searrow \nu\vec{n}.l.e'$ imply that there are f', g' and $\vec{n}'$ such that $e' \equiv \nu\vec{n}'.(f'|g')$. Then for any h, $\nu\vec{m}.(f|h|g) \searrow \nu\vec{n}.l.\nu\vec{n}'.(f'|h|g')$.

Proof

The first part follows directly from lemmas 9.15, 9.16, 9.17 and then 9.9. The second part then follows in a similar way, with additional use of either the PAR$_1$ or PAR$_2$ rule. □

Lemma 9.25

For CML$_\nu$ expressions e, f, g, h and e', label l and names $\vec{n}$ and $\vec{m}$ then $e \equiv \nu\vec{m}.(f|g|h)$ and $e \searrow \nu\vec{n}.l.e'$ imply that there are f', g', h' and $\vec{n}'$ such that $e' \equiv \nu\vec{n}'.(f'|g'|h')$ and $\nu\vec{m}.(g|f|h) \searrow \nu\vec{n}.l.\nu\vec{n}'.(g'|f'|h')$.

Proof

The first part follows from lemmas 9.15, 9.16 and 9.17. The second part follows by piecing together the various bits given as a result of the lemmas used in the first part. □

Lemma 9.26

For CML$_\nu$ expressions e and e', names $\vec{n}$ and value v then $e \searrow \nu\vec{n}.\surd v.e'$ implies that there is an f with $e \equiv \nu\vec{n}.(f|v)$ and $e' \equiv f|\Lambda$.

Proof

This follows easily by induction on the derivation of $e \searrow \nu\vec{n}.\surd v.e'$. □

Lemma 9.27

For CML$_\nu$ expressions e, e' and e'', names $\vec{n}$ and $\vec{m}$, label l and value v then $e \searrow \nu\vec{n}.\surd v.e'$ and $\nu\vec{n}.e' \searrow \nu\vec{m}.l.e''$ implies that there is an e''' with $e \searrow \nu\vec{m}.l.e'''$ and $\nu\vec{m}.e''' \searrow \nu\vec{n}.\surd v.e''$.

Proof

By lemma 9.26 there is an f with $e \equiv \nu\vec{n}.(f|v)$ and $e' \equiv f|\Lambda$. Then by lemmas 9.15 and 9.16 we see that there are g and $\vec{n}'$ with $e'' \equiv \nu\vec{n}'.g|\Lambda$, $\vec{n}' = \vec{n} \cap \vec{m}$, $\vec{n}'' = \vec{n} \setminus \vec{m}$ and $\nu\vec{n}.f \searrow \nu\vec{m}.l.\nu\vec{n}'.g$. We now let $e''' = \nu\vec{n}'.(g|v)$. The result follows using various RES and PAR rules. □

Lemma 9.28

For CML$_\nu$ expressions e and e', names $\vec{n}$ and m and value v then $e \searrow \nu\vec{n}.m!v.e'$ implies that for any names $\vec{n}'$ with $\vec{n}' \cap \mathrm{fn}(e) = \emptyset$ and a 1-1 mapping between $\vec{n}$ and $\vec{n}'$ then $e \searrow \nu\vec{n}'.m!v[\vec{n}'/\vec{n}].e'[\vec{n}'/\vec{n}]$. Similarly if $e \searrow \nu\vec{n}.\surd v.e'$ then $e \searrow \nu\vec{n}'.\surd v[\vec{n}'/\vec{n}].e'[\vec{n}'/\vec{n}]$.

Proof

Firstly we note that $\vec{n}' \cap \mathrm{fn}(e) = \emptyset$ implies that $m \notin \vec{n}'$, $\vec{n}' \cap \mathrm{fn}(\nu\vec{n}.e') = \emptyset$ and $\vec{n}' \cap \mathrm{fn}(\nu\vec{n}.v) = \emptyset$ as well. We consider the (normal) derivation of $e \searrow \nu\vec{n}.m!v.e'$. (By lemma 9.10 such a derivation exists.) We then just relabel all names according to the 1-1 map. Finally we use alpha-conversion to restore bound names in e, v and e' to finalise the result as required. □

9.2 Link between evaluation and transition

We now come to the main result of this chapter. This is the equivalence of evaluation and transition. We first give the main result, and then split the proof into the two directions.

Theorem 9.29

For CML$_\nu$ processes e and e', $e \Downarrow \nu\vec{n}.l.e'$ implies and is implied by $e \searrow \tau^*.\nu\vec{n}.l.e'$.

We give the proof near the end of this section, with most of the section being given over to the lemmas required for the proof.

If we compare theorem 9.29, above, with the corresponding result for NCCS (theorem 5.5 on page 71) then we see the result for CML$_\nu$ is much cleaner. There are two main reasons for this. The first is that we do not have an explicit silent action, τ, in the syntax of CML$_\nu$. Nor do we have an extended syntax that includes silent actions. The second reason is that we also have the (STRUC) rule in the transition relation. In particular, if the unfolding of recursive processes is added to the structural congruence for NCCS, the strong CCS congruence in theorem 5.5 may be replaced by the structural congruence, $\equiv$. We do not have recursive processes in CML$_\nu$, only recursive functions which expand as a silent step in the transition relation, so this problem does not occur in CML$_\nu$.

9.2.1 Evaluation to Transition

We first prove some initial lemmas.

Lemma 9.30

For each evaluation context $\mathcal{E}[-]$ (cf. figure 8.6) $e \searrow \tau.e' \Rightarrow \mathcal{E}[e] \searrow \tau.\mathcal{E}[e']$

Proof

We proceed by induction on the structure of $\mathcal{E}[-]$. In each case if $e \searrow \tau.e'$ then $\mathcal{E}[e] \searrow \tau.\mathcal{E}[e']$ follows using the appropriate reduction rule. □

Lemma 9.31

$e \Downarrow \nu\vec{n}.l.e' \Rightarrow e \searrow \tau^*.\nu\vec{n}.l.e'$

Proof

We proceed by induction on the derivation of $e \Downarrow \nu\vec{n}.l.e'$. Last rule used:

VAL Hence $e = v$. Hence $v \searrow \surd v.\Lambda$.

OUT Hence $e = n!v$ and $n!v \searrow n!v.()$.

IN Hence $e = n?$ and $n? \searrow n?x.x$.

SUM$_1$ Hence $e = ge_1 \oplus ge_2$ and $ge_1 \Downarrow \nu\vec{n}.\alpha.e'$. Hence by induction hypothesis $ge_1 \searrow \tau^*.\nu\vec{n}.\alpha.e'$. Observe that $ge_1 \not\searrow \tau.e''$ because of the restrictions on the syntax of guarded expressions. Hence $ge_1 \searrow \nu\vec{n}.\alpha.e'$ and $e \searrow \nu\vec{n}.\alpha.e'$ using SUM$_1$.

SUM$_2$ Follows in the same way as SUM$_1$.

WRAP Hence $e = ge \Rightarrow v$, $ge \Downarrow \nu\vec{n}.\alpha.e''$ and $e \Downarrow \nu\vec{n}.\alpha.ve''$. Then by induction hypothesis and the same argument as in SUM$_1$ $ge \searrow \nu\vec{n}.\alpha.e''$ and hence using WRAP, $e \searrow \nu\vec{n}.\alpha.e'$.

STRUC Hence $e \equiv f$, $f \Downarrow \nu\vec{n}.l.f'$, and $f' \equiv e'$. Hence by induction hypothesis $f \searrow \tau^*.\nu\vec{n}.l.f'$. But $\exists f_i$ s.t. $f = f_0$, $f_i \searrow \tau.f_{i+1}$ and $f_n \searrow \nu\vec{n}.l.f'$. Then applying STRUC to the first and last in the sequence gives $e \searrow \tau^*.\nu\vec{n}.l.e'$ as required.

RES (of all types) Hence $e = \nu m.f$. By induction hypothesis $f \searrow \tau^*.\nu\vec{n}.l.f'$ where f' is e' without the added restriction. Defining f_i as above, we get that applying RES$_\tau$ to $f_i \searrow \tau.f_{i+1}$ for $i < n$ and the corresponding RES when $i = n$ gives $e \searrow \tau^*.\nu\vec{n}.l.e'$ as required.

PAR$_1$ Hence $e = f|e_2$, $f \Downarrow \nu\vec{n}.l.f'$ and $e' = f'|e_2$. By induction hypothesis $f \searrow \tau^*.\nu\vec{n}.l.f'$. We define f_i as above. Then for $i < n$, $f_i \searrow \tau.f_{i+1}$ implies that $f_i|e_2 \searrow \tau.(f_{i+1}|e_2)$ using PAR$_1$. Also $f_n \searrow \nu\vec{n}.l.f'$ implies that $f_n|e_2 \searrow \nu\vec{n}.l.(f'|e_2)$ and hence we have $e \searrow \tau^*.\nu\vec{n}.l.e'$ as required.

PAR$_2$ follows in a similar fashion to PAR$_1$.

CON$_\alpha$ follows in a similar fashion to PAR$_1$.

APP$_\alpha$ follows in a similar fashion to PAR$_1$.

PAIR$_\alpha$ follows in a similar fashion to PAR$_1$.

IF$_\alpha$ follows in a similar fashion to PAR$_1$.

LET$_\alpha$ follows in a similar fashion to PAR$_1$.

CON$_\surd$ Hence $e = \mathcal{E}[cf]$, $f \Downarrow \nu\vec{m}.\surd v.f'$ and $\mathcal{E}[\nu\vec{m}.(f'|\delta(c,v))] \Downarrow \nu\vec{n}.l.e'$. Therefore, using the induction hypothesis we get $f \searrow \tau^*.\nu\vec{m}.\surd v.f'$ and $\mathcal{E}[\nu\vec{m}.(f'|\delta(c,v))] \searrow \tau^*.\nu\vec{n}.l.e'$ Again we define f_i as above. Then for $i < n$ we use CON$_\alpha$ and lemma 9.30 to get that $\mathcal{E}[cf_i] \searrow \tau.\mathcal{E}[cf_{i+1}]$. Then $f_n \searrow \nu\vec{m}.\surd v.f'$ implies that $cf_n \searrow \tau.\nu\vec{m}.(f'|\delta(c,v))$ and then applying lemma 9.30 to this gives $\mathcal{E}[cf_n] \searrow \tau.\mathcal{E}[\nu\vec{m}.(f'|\delta(c,v))]$. Putting these all together we get $e \searrow \tau^*.\nu\vec{n}.l.e'$ as required.

IF$_t$ follows in the same way as CON$_\surd$.

IF$_f$ follows in the same way as CON$_\surd$.

PAIR$_\surd$ follows in the same way as CON$_\surd$.

APP$_\surd$ follows in the same way as CON$_\surd$.

LET$_\surd$ follows in the same way as CON$_\surd$.

COM$_1$ Hence $e = \mathcal{E}[f_1|f_2]$ and there are f_1', f_2', m', $\vec{m}$ and v with $f_2 \Downarrow m'?x.f_2'$, $f_1 \Downarrow \nu\vec{m}.m'!v.f_1'$, and $\mathcal{E}[\nu\vec{m}.(f_1'|f_2'[v/x])] \Downarrow \nu\vec{n}.l.e'$. By induction hypothesis $f_1 \searrow \tau^*.\nu\vec{m}.m'!v.f_1'$ and $f_2 \searrow \tau^*.m'?x.f_2'$. We define f_{1i} and f_{2j} in the obvious manner with $f_{1n} \searrow \nu\vec{m}.m'!v.f_1'$ and $f_{2m} \searrow m'?x.f_2'$. Using PAR$_2$ and lemma 9.30 (multiple times) we get $\mathcal{E}[f_1|f_2] \searrow \tau^*.\mathcal{E}[f_{1n}|f_2]$ and then using PAR$_2$ and lemma 9.30 (multiple times) we get $\mathcal{E}[f_{1n}'|f_2] \searrow \tau^*\mathcal{E}[f_{1n}|f_{2m}]$. Then using COM$_1$ and lemma 9.30 gives $\mathcal{E}[f_{1n}|f_{2m}] \searrow \tau\mathcal{E}[\nu\vec{m}.(f_1'|f_2'[v/x])]$. Then induction hypothesis on $\mathcal{E}[\nu\vec{m}.(f_1'|f_2'[v/x])] \Downarrow \nu\vec{n}.l.e'$ gives $\mathcal{E}[\nu\vec{m}.(f_1'|f_2'[v/x])] \searrow \tau^*\nu\vec{n}.l.e'$. Putting these all together gives the required result.

COM$_2$ Similar to COM$_1$.

□

9.2.2 Transition to Evaluation

Lemma 9.32

$e \searrow \nu\vec{n}.l.e' \;\&\; l \neq \tau \Rightarrow e \Downarrow \nu\vec{n}.l.e'$

Proof

The derivation of $e \searrow \nu\vec{n}.l.e'$ gives a derivation of $e \Downarrow \nu\vec{n}.l.e'$ using the same rules, observing that once a τ transition is introduced it remains. □

Lemma 9.33

$e \searrow \tau.e' \;\&\; \mathcal{E}[e'] \Downarrow C \Rightarrow \mathcal{E}[e] \Downarrow C$

Proof

We proceed by induction on the derivation of $e \searrow \tau.e'$. Last rule used:

VAL Cannot occur since $\sqrt{v} \neq \tau$.

OUT Similarly cannot occur.

IN Cannot occur.

SUM$_1$ Cannot occur since $ge \not\searrow \tau.e'$.

SUM$_2$ Similarly cannot occur.

WRAP Cannot occur for same reason as above.

STRUC Hence we have $e \equiv f$, $f \searrow \tau.f'$ and $f' \equiv e'$. $\mathcal{E}[e'] \equiv \mathcal{E}[f']$ implies that $\mathcal{E}[f'] \Downarrow C$ using STRUC. Hence by induction hypothesis $\mathcal{E}[f] \Downarrow C$. Then $e \equiv f$ implies that $\mathcal{E}[e] \equiv \mathcal{E}[f]$ and hence using STRUC again we get $\mathcal{E}[e] \Downarrow C$ as required.

RES$_\tau$ (N.B. this is the only possible RES rule that can occur.) Hence we have $e = \nu n.f$ and $e' = \nu n.f'$. $\mathcal{E}[\nu n.f']$ is an evaluation context and hence by induction hypothesis $\mathcal{E}[\nu n.f] \Downarrow C$ as required.

PAR$_1$ Follows in a similar manner to RES$_\tau$.

PAR$_2$ Follows in a similar manner to RES$_\tau$.

CON$_\alpha$ Follows in a similar manner to RES$_\tau$.

APP$_\alpha$ Follows in a similar manner to RES$_\tau$.

PAIR$_\alpha$ Follows in a similar manner to RES$_\tau$.

IF$_\alpha$ Follows in a similar manner to RES$_\tau$.

LET$_\alpha$ Follows in a similar manner to RES$_\tau$.

CON$_\surd$ Hence we have $e = cf$, $e' = \nu\vec{n}.(f'|\delta(c,v))$, $f \searrow \nu\vec{n}.\surd v.f'$ and $\mathcal{E}[\nu\vec{n}.(f'|\delta(c,v))] \Downarrow C$. By lemma 9.32 $f \Downarrow \nu\vec{n}.\surd v.f'$ and hence using CON$_\surd$ we get $e = cf \Downarrow C$ as required.

IF$_t$ Follows in similar fashion to CON$_\surd$.

IF$_f$ Follows in similar fashion to CON$_\surd$.

PAIR$_\surd$ Follows in similar fashion to CON$_\surd$.

APP$_\surd$ Follows in similar fashion to CON$_\surd$.

LET$_\surd$ Follows in similar fashion to CON$_\surd$.

COM$_1$ Hence we have $f_1 \searrow \nu\vec{m}.n!v.f_1'$, $f_2 \searrow n?x.f_2'$, $e = f_1|f_2$ and $\mathcal{E}[\nu\vec{m}.(f_1'|f_2'[v/x])] \Downarrow C$. Then using lemma 9.32 twice and COM$_1$ gives $e \Downarrow C$ as required.

COM$_2$ Follows in similar fashion to COM$_1$.

□

Lemma 9.34

$e \searrow \tau^*.\nu\vec{n}.l.e' \Rightarrow e \Downarrow \nu\vec{n}.l.e'$

Proof

Follows from lemmas 9.32 and 9.33 using induction on the number of τ steps. □

Restatement of Theorem 9.29

For CML$_\nu$ processes e and e', $e \Downarrow \nu\vec{n}.l.e'$ implies and is implied by $e \searrow \tau^*.\nu\vec{n}.l.e'$.

Proof

This follows from lemmas 9.31 and 9.34. □

Corollary 9.35

For CML$_\nu$ expressions e, f and g, names $\vec{n}$ and $\vec{m}$, value v and label l then $e \Downarrow \nu\vec{n}.\surd v.f$ and $\nu\vec{n}.f \Downarrow \nu\vec{m}.l.g$ imply that there is an h with $e \Downarrow \nu\vec{m}.l.h$ and $\nu\vec{m}.h \searrow \nu\vec{n}.\surd v.g$.

Proof

$e \Downarrow \nu\vec{n}.\surd v.f$ implies that there are $e_0, e_1, \ldots, e_p$ with $e = e_0$, $e_p \searrow \nu\vec{n}.\surd v.f$ and $e_i \searrow \tau.e_{i+1}$. Similarly there are $f_0, f_1, \ldots, f_q$ with $f = f_0$, $f_q \searrow \nu\vec{m}.l.g$ and $f_i \searrow \tau.f_{i+1}$. The result follows by applying lemma 9.27 $q + 1$ times. □

Chapter 10

Bisimulation on CML_ν

We now turn our attention to the bisimulations we intend to give for CML_ν. This is a fairly long and mixed chapter so we give an outline of its layout.

10.1 Outline of the Chapter

The main content of the chapter starts in section 10.2 with a discussion of the various aspirations we may have for any equivalences we define on CML_ν. We then have a look at various simple expressions and consider whether we would like them to be equivalent. This is in section 10.3. In the following section we consider a particular pair of expressions which give us some deeper understanding of some of the issues involved in deciding whether two expressions should be equivalent. This also allows us to have a look at the issue of privacy of channel names. One idea that sometimes helps us to understand the concept of dynamic channel creation is that such a channel is "private" and only accessible to those expressions to which this "private" channel name is passed. We consider how "private" a dynamically created channel name is.

In section 10.5 we define *Evaluation Bisimulation*, our main, and weakest, equivalence relation for CML_ν. We then go on in section 10.6 to define *Strong Bisimulation* for CML_ν and continue by giving a reduced version of Strong Bisimulation. This reduced version is then proved to be equivalent to the original version. In section 10.7 we prove various useful properties of Strong Bisimulation. These properties will be used to prove

the equivalent properties for Evaluation Bisimulation and to mirror all the results given in the section on properties of Evaluation Bisimulation that will be used in the proof of congruence for Evaluation Bisimulation. We then give the corresponding proofs for Evaluation Bisimulation in section 10.8. The chapter concludes with two final sections. The first gives some basic examples of equivalent expressions and the second describes the relationship of our equivalences with those given in [FHJ95].

10.2 Aspirations for an Equivalence

In section 8.1 we mentioned that Reppy did not define an equivalence between processes. Having defined CML$_\nu$ we now turn our attention to defining an equivalence for expressions in CML$_\nu$. We first consider what properties we want our equivalence to satisfy.

1. The first criterion that we desire is that the equivalence is closed under reduction, in particular under evaluation. If we wish to replace one expression with another, possibly under restricted circumstances, we do not want the replacement to cease to behave the same way after just one evaluation step. We wish it to be able to mimic the other forever.

2. A second criterion is that we wish to replace one expression by another that does not just mimic the other, but also does no more than the expression it is replacing. It is therefore desirable that the two expressions be completely interchangeable, i.e. we wish to have a bisimulation and not just a simulation.

3. Thirdly we would like bisimilar processes to be exchangeable in as many contexts as possible. Ideally we want the bisimulation to be a congruence relation.

4. We also wish the bisimulation to be an equivalence relation, and in particular it should be reflexive. Since there is non-determinism in the language, we only require that one expression *may* mimic another, and not that it *must*[1] mimic the other.

[1]For a more detailed discussion of may and must see, for example, [Hen94].

5. Finally we would like the equivalence to be as weak as possible without allowing the two expressions to be observably different. This leads us to another question, which is "what do we consider to be observable?" We take the following decisions about what we may observe:

   (a) The returning of a value is observable. However it is less obvious whether the value itself should be observable. In particular, if the value is restricted, that is it has restricted channel names as part of it, then the values possible (in)equivalence with another value, which may also have restricted names, may depend on the expressions which returned them as values. We give some examples of this in section 10.3. However, just ignoring the returned value is too weak. Therefore we allow the value to be applied to a function in the context of the returning expression.

   (b) An expression that commits to transmitting a value on a channel is also taken to be observable. The channel upon which the value is transmitted is also observable. However, for the same reasons given above, the value is only observable by being applied to a function in the context of the returning expression.

   (c) An expression committing to receiving on a specified, unrestricted channel is regarded as being observable. Indeed we will view it as being the case in which we may not only know the name of the channel, but we may also "reply" to the "request" for an input, and send a value on the channel to answer the "request".

   (d) A further question is whether any silent actions should be observable. We might regard all actions as taking a fixed length of time and so we can observe silent actions by watching a clock. However the very name "silent actions" suggests that they should not be observable. Indeed this is part of the background to evaluation. We could well argue that a practical use of equivalence is to replace one slow process by another faster one. The main difference may be in terms of the number of silent actions that each performs.

10.3 Equivalent or not?

We now present various pairs of expressions which we may wish to be equivalent or not. The first pair to be considered are:

$$v|\Lambda \overset{?}{\cong} \Lambda$$

for any value v. At first it may seem that we do not want these two to be equivalent, however neither can do any evaluation or transition.

We now consider another case:

$$\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ n?\ \mathbf{then\ true\ else\ false})$$
$$\overset{?}{\cong}$$
$$\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ n?\ \mathbf{then\ false\ else\ true})$$

These are two expressions which we intuitively say should not be equal. This is because if they are sent either **true** or **false**, having been applied to any value, then they each give a result. However the results will be different. We now consider a modified version:

$$A = \nu n.(n!\mathbf{true}|\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ n?\ \mathbf{then\ true\ else\ false}))$$
$$\overset{?}{\cong}$$
$$B = \nu n.(n!\mathbf{false}|\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ n?\ \mathbf{then\ false\ else\ true}))$$

We now have the situation where each expression will give a result which is a restricted value. The unrestricted versions of the values are not equivalent, as noted above. However this does not necessarily mean that they are inequivalent. We might ask whether there is a CML$_\nu$ function which will return **true** if given expression A and **false** if given B. This function will take the function returned by A (or B) as an argument, and then may create as many copies as it wishes. However each will have the channel name n restricted. Each copy would use the same n though. However there can only be one copy of $n!\mathbf{true}$ (or $n!$**false**). This may suggest that the two are equivalent. It certainly suggests that restriction complicates the question of equivalence.

10.4 Equivalence — extended problem

We will consider a version of an example used by Stark in [Sta95] to show how hard it is to give a precise notion of privacy. We define two expressions F and F' below:

$$F \triangleq \nu y.\nu z.(\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ (\mathbf{match} < x, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y))$$
$$F' \triangleq \nu y.\nu z.(\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ (\mathbf{match} < x, z >)\ \mathbf{then}\ z\ \mathbf{else}\ y))$$

The idea is to consider how private "private names" are. If the names y and z were completely private in F and F' then the two expressions should be equivalent. Therefore the study of these two expressions gives us some indication about how strong privacy is.

The main question is whether F and F' should be thought of as equivalent. Both F and F' can evaluate to functions, which are values. These functions each have two restricted, or private, names y and z. However, for the moment we will give an example of an evaluation for F when it has been applied to a name, n. We recall that names are also values.

$$\mathrm{A} = \dfrac{\dfrac{\dfrac{}{< n, y >\Downarrow \surd < n, y > .\Lambda} \quad \dfrac{\dfrac{}{\mathbf{false} \Downarrow \surd \mathbf{false}.\Lambda}}{\Lambda|\mathbf{false} \Downarrow \surd \mathbf{false}.\Lambda}}{\mathbf{match} < n, y >\Downarrow \surd \mathbf{false}.\Lambda} \quad \dfrac{\dfrac{}{y \Downarrow \surd y.\Lambda}}{\Lambda|y \Downarrow \surd y.\Lambda}}{\mathbf{if}\ (\mathbf{match} < n, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y \Downarrow \surd y.\Lambda}$$

$$\mathrm{B} = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{n \Downarrow \surd n.\Lambda \quad \dfrac{\mathrm{A}}{\Lambda|\mathbf{if}\ (\mathbf{match} < n, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y \Downarrow \surd y.\Lambda}}{\mathbf{let}\ u{=}n\ \mathbf{in}\ (\mathbf{if}\ (\mathbf{match} < u, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y) \Downarrow \surd y.\Lambda}}{\nu z.(\mathbf{let}\ u{=}n\ \mathbf{in}\ (\mathbf{if}\ (\mathbf{match} < u, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y)) \Downarrow \surd y.\nu z.\Lambda}}{\nu y.\nu z.(\mathbf{let}\ u{=}n\ \mathbf{in}\ (\mathbf{if}\ (\mathbf{match} < u, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y)) \Downarrow \nu y.\surd y.\nu z.\Lambda}}{\nu y.\nu z.(\Lambda|\mathbf{let}\ u{=}n\ \mathbf{in}\ (\mathbf{if}\ (\mathbf{match} < u, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y)) \Downarrow \nu y.\surd y.\Lambda}$$

$$\dfrac{F \Downarrow \nu y.\nu z.\surd \mathbf{fix}\,(v{=}\mathbf{fn}x{\Rightarrow}(\mathbf{if}\ (\mathbf{match} < x, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y))\,.\Lambda \quad \mathrm{B}}{Fn \Downarrow \nu y.\surd y.\Lambda}$$

Figure 10.1: Example function evaluation

The derivation for $Fn \Downarrow \nu y.\surd y.\Lambda$ is given in figure 10.1. We may observe that the function F is deterministic since it does not have any communications and, although it accepts any value, it only evaluates if the value it is applied to is a name. We may see in a similar manner that $F'n$ evaluates to $\nu y.\surd y.\Lambda$ as well. This may lead us to think that F and F' are equivalent. They can both be applied to a name, and only evaluate when the value they are applied to is a name. When they are applied to a name they both return the same result, noting that both y and z are private names.

However this suggestion leads us to the wrong conclusion. The following expression returns a different result when applied to F and F':

$$G = \mathbf{fn}\,(f \Rightarrow [\mathbf{let}\ u = fn\ \mathbf{in}\ (\mathbf{let}\ v = fu\ \mathbf{in}\ [\mathbf{match} < u, v >])])$$

We will write F_1 for $\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ (\mathbf{match} < x, y >)\ \mathbf{then}\ z\ \mathbf{else}\ y)$ and F_1' for $\mathbf{fn}\,(x \Rightarrow \mathbf{if}\ (\mathbf{match} < x, z >)\ \mathbf{then}\ z\ \mathbf{else}\ y)$. We may then write F as $\nu y.\nu z.F_1$ and F' as $\nu y.\nu z.F_1'$. Using a derivation very similar to the one in figure 10.1 we can show that $F_1 n \Downarrow \surd y.\Lambda$. Similar derivations also show the following:

- $F_1' n \Downarrow \surd y.\Lambda$
- $F_1 y \Downarrow \surd z.\Lambda$
- $F_1' y \Downarrow \surd y.\Lambda$

We give the derivation trees for $GF \Downarrow \surd\mathbf{false}.\Lambda$ and $GF' \Downarrow \surd\mathbf{true}.\Lambda$ in figures 10.2 and 10.3 respectively. From this we may see that it is possible to differentiate between F and F'. This is in accord with [Sta95]. There is a requirement to be allowed to use higher order functions in order to be able to show the difference between F and F'. The bisimulations we define in sections 10.5 and 10.6 can both show that F and F' differ.

Therefore, returning to our original question about how private "private names" are, we see that private names are not perfectly private. Any expression which wishes to have a private name must itself ensure that it does not allow it to be given away to another expression. We see, just as Stark did in [Sta95], that privacy is a very difficult concept to quantify.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\textbf{false} \Downarrow \surd\textbf{false}.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{false}) \Downarrow \surd\textbf{false}.\Lambda}}{\dfrac{< y, z > \Downarrow \surd < y, z > .\Lambda}{\nu y.\nu z.\textbf{match} < y, z > \Downarrow \surd\textbf{false}.\Lambda}}}{F_1 y \Downarrow \surd z.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{match} < y, z >) \Downarrow \surd\textbf{false}.\Lambda}}{\nu y.\nu z.(\textbf{let } v = F_1 y \textbf{ in } [\textbf{match} < y, v >]) \Downarrow \surd\textbf{false}.\Lambda}}{F_1 n \Downarrow \surd y.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{let } v = F_1 y \textbf{ in } [\textbf{match} < y, v >]) \Downarrow \surd\textbf{false}.\Lambda}}{\nu y.\nu z.(\textbf{let } u = F_1 n \textbf{ in } (\textbf{let } v = F_1 u \textbf{ in } [\textbf{match} < u, v >])) \Downarrow \surd\textbf{false}.\Lambda}}{F \Downarrow \nu y.\nu z.F_1.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{let } u = F_1 n \textbf{ in } (\textbf{let } v = F_1 u \textbf{ in } [\textbf{match} < u, v >])) \Downarrow \surd\textbf{false}.\Lambda}}{\nu y.\nu z.(\textbf{let } f = F_1 \textbf{ in } [\textbf{let } u = f n \textbf{ in } (\textbf{let } v = f u \textbf{ in } [\textbf{match} < u, v >])]) \Downarrow \surd\textbf{false}.\Lambda}}{G \Downarrow \surd G.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{let } f = F_1 \textbf{ in } [\textbf{let } u = f n \textbf{ in } (\textbf{let } v = f u \textbf{ in } [\textbf{match} < u, v >])]) \Downarrow \surd\textbf{false}.\Lambda}$$

$$GF \Downarrow \surd\textbf{false}.\Lambda$$

Figure 10.2: Evaluation derivation for GF

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{G \Downarrow \surd G.\Lambda} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F' \Downarrow \nu y.\nu z.F_1'.\Lambda} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F_1' n \Downarrow \surd y.\Lambda} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F_1' y \Downarrow \surd y.\Lambda} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{< y,y > \Downarrow \surd < y,y > .\Lambda} \quad \dfrac{}{\textbf{true} \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{true}) \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.\textbf{match} < y,y > \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{match} < y,y >) \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\textbf{let } v = F_1' y \textbf{ in } [\textbf{match} < y,v >]) \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{let } v = F_1' y \textbf{ in } [\textbf{match} < y,v >]) \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\textbf{let } u = F_1' n \textbf{ in } (\textbf{let } v = F_1' u \textbf{ in } [\textbf{match} < u,v >])) \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{let } u = F_1' n \textbf{ in } (\textbf{let } v = F_1' u \textbf{ in } [\textbf{match} < u,v >])) \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\textbf{let } F' = F_1' \textbf{ in } [\textbf{let } u = F' n \textbf{ in } (\textbf{let } v = F' u \textbf{ in } [\textbf{match} < u,v >])]) \Downarrow \surd \textbf{true}.\Lambda}}{\nu y.\nu z.(\Lambda|\textbf{let } F' = F_1' \textbf{ in } [\textbf{let } u = F' n \textbf{ in } (\textbf{let } v = F' u \textbf{ in } [\textbf{match} < u,v >])]) \Downarrow \surd \textbf{true}.\Lambda}}{GF' \Downarrow \surd \textbf{true}.\Lambda}$$

Figure 10.3: Evaluation derivation for GF'

10.5 Evaluation Bisimulation for CML$_\nu$

We follow Sangiorgi [San92] in generalising Milner's ideas of *concretion* and *abstraction* in [Mil91]. We let concretions be of the form $\nu\vec{n} < v, e >$, and abstractions be (x, e), where $\vec{n}$ is a, possibly empty, set of channel names, v is a value, x is a variable and e is an expression. We also require that $\vec{n} \subseteq \text{fn}(v)$.

Notation 10.1

We let C, D represent concretions and A, B represent abstractions. Then, if C is $\nu\vec{n} <v, e'>$ and A is (x, e'), we write $e \Downarrow m!C$ as shorthand for $e \Downarrow \nu\vec{n}.m!v.e'$, $e \Downarrow \surd C$ for $e \Downarrow \nu\vec{n}.\surd v.e'$ and $e \Downarrow m?A$ for $e \Downarrow m?x.e'$.

Notation 10.2

We write CML$_\nu$ to mean the set of closed CML$_\nu$ expressions where the context makes it clear that we are talking about expressions rather than the name of the calculus.

Definition 10.3

We also extend Milner's *pseudo-application* [Mil91]. If $A = (x, e)$ and $C = \nu\vec{n}<v, f>$ then using α-conversion to ensure that $x \notin \vec{n}$ and $\text{fn}(e) \cap \vec{n} = \emptyset$ we define

$$A \bullet C = \nu\vec{n}.(e[v/x]|f)$$

$$C \bullet A = \nu\vec{n}.(f|e[v/x])$$

We may now define Evaluation Bisimulation. We recall from section 8.4 that the rightmost part of a parallel expression is special, in that it is the only part that may return a value. All other parts may only receive or transmit values. When we defined Evaluation Bisimilarity for NCCS in section 3.1 on page 42 we restricted the contexts used to distinguish between inequivalent expressions. In particular we restricted the contexts to be parallel NCCS expression. In the CML$_\nu$ case we might think that in order to have our equivalence being a congruence relation we would require the contexts to be more complicated. In fact we will define our bisimulation using only contexts which have a single CML$_\nu$ expression added in parallel, and furthermore it will be added on the left. In the NCCS case we merely required that the continuations of the two expressions remained equivalent.

As we have already discussed in section 10.2 this would be too weak for CML$_\nu$. Instead we allow returned or transmitted values to be received by a CML$_\nu$ expression, and require that the expressions in parallel with the corresponding continuations continue to be equivalent.

We will see in corollary 10.22 that the order of expressions in a parallel composition doesn't matter as long as the rightmost expression remains as the rightmost one. This means that we may view a parallel composition of CML$_\nu$ expressions as if the rightmost one is the "controlling" expression which interacts with the user by returing a value. The other expressions are like Unix background processes, or DOS TSR[2] programs. The bisimulation merely adds more "background processes". This may appear to be quite weak, however we prove in Theorem 10.5 (using chapters 11 and 12) that Evaluation Bisimulation is indeed a Congruence relation.

Definition 10.4 *Evaluation Bisimulation for CML$_\nu$*

Given $\mathcal{R} \subseteq \text{CML}_\nu \times \text{CML}_\nu$, define $e_1[\mathcal{R}]e_2$ to hold if and only if for any f, a closed CML$_\nu$ expression, and n, a channel name, then:

$$\begin{array}{rcl}
f|e_1 \Downarrow n!C & \Rightarrow & \exists D\,[f|e_2 \Downarrow n!D \quad \& \quad \forall A\ (A \bullet C)\mathcal{R}(A \bullet D) \\
\& \quad f|e_1 \Downarrow \surd C & \Rightarrow & \exists D\,[f|e_2 \Downarrow \surd D \quad \& \quad \forall A\ (A \bullet C)\mathcal{R}(A \bullet D) \\
\& \quad f|e_1 \Downarrow n?A & \Rightarrow & \exists B\,[f|e_2 \Downarrow n?B \quad \& \quad \forall C\ (C \bullet A)\mathcal{R}(C \bullet B)
\end{array}$$

$$\begin{array}{rcl}
\& \quad f|e_2 \Downarrow n!D & \Rightarrow & \exists C\,[f|e_1 \Downarrow n!C \quad \& \quad \forall A\ (A \bullet C)\mathcal{R}(A \bullet D) \\
\& \quad f|e_2 \Downarrow \surd D & \Rightarrow & \exists C\,[f|e_1 \Downarrow \surd C \quad \& \quad \forall A\ (A \bullet C)\mathcal{R}(A \bullet D) \\
\& \quad f|e_2 \Downarrow n?B & \Rightarrow & \exists A\,[f|e_1 \Downarrow n?A \quad \& \quad \forall C\ (C \bullet A)\mathcal{R}(C \bullet B)
\end{array}$$

Then we let $\simeq$ be $\nu\mathcal{R}.[\mathcal{R}]$. We let $\simeq^o$ be the extension to open processes by substitution of closed values for free variables.

Theorem 10.5

Evaluation Bisimulation is a congruence relation.

Proof

We split this into two parts. Firstly we observe that Evaluation Bisimulation is an equivalence relation following the same logic as theorem

[2]Terminal Stay Resident.

3.7(1). We prove the other (compatibility) properties of congruence in chapter 12.

□

10.6 Strong Bisimulation for CML$_\nu$

Before we go on to proving properties about Evaluation Bisimulation we will define Strong Bisimulation. This will be useful as a tool in proving various properties of Evaluation Bisimulation.

Here we define Strong Bisimulation, together with a reduced version. We then show that the two versions are equivalent. The two versions can then be used interchangebly, which will simplify some of the proofs of the properties given in the next section.

The definition of Strong Bisimulation corresponds almost exactly to the definition of Evaluation Bisimulation (definition 10.4 on the previous page) with evaluation being replaced by transition. There is also an extra condition added when there is a τ action. The same considerations about symmetry apply as for Evaluation Bisimulation and again we will show that the definition is not too weak.

Definition 10.6 *Strong Bisimulation for CML$_\nu$*

Given $\mathcal{R} \subseteq \text{CML}_\nu \times \text{CML}_\nu$, define $e_1\{\mathcal{R}\}e_2$ to hold if and only if for any f, a closed CML$_\nu$ expression, and n, a channel name, then:

$$
\begin{array}{lllll}
 & f|e_1 \searrow n!C \Rightarrow \exists D\,[f|e_2 \searrow n!D & \& & \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & f|e_1 \searrow \surd C \Rightarrow \exists D\,[f|e_2 \searrow \surd D & \& & \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & f|e_1 \searrow n?A \Rightarrow \exists B\,[f|e_2 \searrow n?B & \& & \forall C\,(C \bullet A)\mathcal{R}(C \bullet B)] \\
\& & f|e_1 \searrow \tau.e_1' \Rightarrow \exists e_2'\,[f|e_2 \searrow \tau.e_2' & \& & e_1'\mathcal{R}e_2'] \\
 & & & & \\
\& & f|e_2 \searrow n!C \Rightarrow \exists D\,[f|e_1 \searrow n!D & \& & \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & f|e_2 \searrow \surd C \Rightarrow \exists D\,[f|e_1 \searrow \surd D & \& & \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & f|e_2 \searrow n?A \Rightarrow \exists B\,[f|e_1 \searrow n?B & \& & \forall C\,(C \bullet A)\mathcal{R}(C \bullet B)] \\
\& & f|e_2 \searrow \tau.e_2' \Rightarrow \exists e_1'\,[f|e_1 \searrow \tau.e_1' & \& & e_1'\mathcal{R}e_2']
\end{array}
$$

Then we let $\sim$ be $\nu\mathcal{R}.\{\mathcal{R}\}$. We let $\sim^o$ be the extension to open processes by substitution of closed values for free variables.

Theorem 10.7

Strong Bisimulation is a congruence relation.

Proof

We split this into two parts. Again we observe that Strong Bisimulation is an equivalence relation following the same logic as theorem 3.7(1). We do not give the proof of the compatibility parts of congruence, but it follows in a corresponding way to the proof of crongruence for Evaluation Bisimulation given in chapter 12. Corresponding versions of all lemmas used in section 10.8 are proved in section 10.7.

□

Definition 10.8 *Reduced Strong Bisimulation for CML$_\nu$*

Given $\mathcal{R} \subseteq \text{CML}_\nu \times \text{CML}_\nu$, define $e_1\{\mathcal{R}\}_r e_2$ to hold if and only if for any n, a channel name, then:

$$\begin{aligned}
& e_1 \searrow n!C \Rightarrow \exists D\,[e_2 \searrow n!D \quad \& \quad \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& \;\; & e_1 \searrow \surd C \Rightarrow \exists D\,[e_2 \searrow \surd D \quad \& \quad \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& \;\; & e_1 \searrow n?A \Rightarrow \exists B\,[e_2 \searrow n?B \quad \& \quad \forall C\,(C \bullet A)\mathcal{R}(C \bullet B)] \\
\& \;\; & e_1 \searrow \tau.e_1' \Rightarrow \exists e_2'\,[e_2 \searrow \tau.e_2' \quad \& \quad e_1'\mathcal{R}e_2']
\end{aligned}$$

$$\begin{aligned}
\& \;\; & e_2 \searrow n!C \Rightarrow \exists D\,[e_1 \searrow n!D \quad \& \quad \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& \;\; & e_2 \searrow \surd C \Rightarrow \exists D\,[e_1 \searrow \surd D \quad \& \quad \forall A\,(A \bullet C)\mathcal{R}(A \bullet D)] \\
\& \;\; & e_2 \searrow n?A \Rightarrow \exists B\,[e_1 \searrow n?B \quad \& \quad \forall C\,(C \bullet A)\mathcal{R}(C \bullet B)] \\
\& \;\; & e_2 \searrow \tau.e_2' \Rightarrow \exists e_1'\,[e_1 \searrow \tau.e_1' \quad \& \quad e_1'\mathcal{R}e_2']
\end{aligned}$$

Then we let $\sim_r$ be $\nu\mathcal{R}.\{\mathcal{R}\}_r$. We let $\sim_r^o$ be the extension to open processes by substitution of closed values for free variables.

Lemma 10.9

$\sim_r^o$ is an equivalence relation, and $e \equiv f$ implies that $e \sim_r^o f$ for any CML$_\nu$ expressions e and f.

Proof

This follows in the same way as the equivalence relation part of Theorem 10.7. □

Lemma 10.10

For any CML$_\nu$ expressions e, f and g then $e \sim_r^o f$ implies that $g|e \sim_r^o g|f$.

Proof

We let $\mathcal{R} = \{(g|e, g|f) \text{ s.t. } e \sim_r f\}$, and we need to show that $\mathcal{R}$ is closed under $\{-\}_r$. Then if $(g|e, g|f) \in \mathcal{R}$ we case split on what trasition $g|e$ performs.

If $g|e \searrow n?x.e'$ for some n and e' then by lemma 9.16 either $g \searrow n?x.g'$ for some g' with $e' \equiv g'|e$ or $e \searrow n?x.g'$ and $e' \equiv g|g'$. In the first case PAR$_2$ implies that $g|f \searrow n?x.(g'|f)$ and for any concretion C we see that $(C \bullet (x, g')|e, C \bullet (x, g')|f) \in \mathcal{R}$. Therefore $(C \bullet (x, g'|e), C \bullet (x, g'|f)) \in \mathcal{R}$ for any C. Otherwise $e \sim_r f$ implies that $f \searrow n?x.f'$ and for any concretion C, $(C \bullet (x, e'), C \bullet (x, f')) \in \mathcal{R}$. Thus if $C = \nu\vec{m}. < v, h >$ then letting $C' = \nu\vec{m}. < v, h|g >$[3] we see that $(C' \bullet (x, e'), C' \bullet (x, f')) \in \mathcal{R}$ which shows that $(C \bullet (x, g|e'), C \bullet (x, g|f')) \in \mathcal{R}$, as required.

If $g|e \searrow \nu\vec{n}.m!v.e'$ for some $\vec{n}$, m, v and e' then by lemma 9.16 either $g \searrow \nu\vec{n}.m!v.g'$ for some g' with $e' \equiv g'|e$ or $e \searrow \nu\vec{n}.m!v.g'$ for some g' with $e' \equiv g|g'$. In each case the same reasoning as used above follows.

If $g|e \searrow \nu\vec{n}.\surd v.e'$ for some $\vec{n}$, v and e' then by lemma 9.16 $e \searrow \nu\vec{n}.\surd v.g'$ for some g' with $e' \equiv g|g'$. The same reasoning as above still follows.

If $g|e \searrow \tau.e'$ for some e' then by lemma 9.17 there are four possibilities. If $e \searrow \tau.e''$ with $e' \equiv g|e''$ then $f \searrow \tau.f''$, for some f'', and $e \sim_r f$. But then $(g|e'', g|f'') \in \mathcal{R}$, as required. If $g \searrow \tau.g'$ and $e' \equiv g'|e$ then $(g'|e, g'|f) \in \mathcal{R}$), as required. If $e \searrow n?x.e'$ and $g \searrow \nu\vec{m}.n!v.g'$ for some n, x, e', $\vec{m}$, v and g' then $f \searrow n?x.f'$ for some f' and for any C we see that $C \bullet (x, e') \sim_r C \bullet (x, f')$. In particular $C = \nu\vec{m} < v, g' >$ and so $\Lambda|\nu\vec{m}.(g'|e'[v/x]) \sim_r \Lambda|\nu\vec{m}.(g'|f'[v/x])$. Therefore $(\nu\vec{m}.(g'|e'[v/x]), \nu\vec{m}.(g'|f'[v/x])) \in \mathcal{R}$, as required. The last case where $g \searrow n?x.g'$ and $e \searrow \nu\vec{m}.n!v.e'$ for some n, x, e', $\vec{m}$, v and g' follows in the same way.

The symmetric cases all follow in the same way. Therefore $\mathcal{R}$ is closed under $\{-\}_r$ and so $\mathcal{R} \subseteq \sim_r^o$, as desired. □

Corollary 10.11

$\sim^o$ and $\sim_r^o$ coincide.

Proof

Follows directly from lemma 10.10. □

[3] α-converting if necessary.

Therefore we may freely use either $\sim^o$ or $\sim^o_\tau$ interchangeably. This will be of much use in the proof of various lemmas.

10.7 Properties of Strong Bisimulation

This section starts with a couple of simple properties. These are that Evaluation Bisimulation is strictly weaker than Strong Bisimulation and that, upto Strong Bisimulation, we may swap expressions in a parallel composition apart from the rightmost expression.

We then consider how the left-hand part of a parallel composition may be pushed in or pulled out in various contexts, again upto Strong Bisimulation. This will be used in the proof of the compatibility parts of congruence, and also in proving the corresponding lemma for Evaluation Bisimulation.

Next we give a slightly surprising lemma (lemma 10.17). We show that for any CML$_\nu$ expressions e, f and g that $e \sim f$ implies that $e|g \sim f|g$. This is surprising because the rightmost expression in a composition is special and we do not introduce a new expression on the right anywhere in the definition of Strong Bisimulation.

The section then concludes with some specific consequences of lemma 10.17 which will be used in the proof of the compatibility parts of congruence for Strong Bisimulation.

Lemma 10.12

For CML$_\nu$ processes e and f then $e \sim^o f$ implies that $e \simeq^o f$.

Proof

This follows from the definitions of the two bisimulations (definitions 10.4 and 10.6) and from theorem 9.29. □

We now show that, upto Strong Bisimulation, we may swap the order of parallel compositions as long as we leave the rightmost expression alone.

Lemma 10.13

$\nu\vec{n}.(e_1|e_2|e_3) \sim^o \nu\vec{n}.(e_2|e_1|e_3)$

Proof

This follows directly from lemma 9.25. □

Corollary 10.14

Let $\prod e_1 = e_1|e_2|\ldots|e_n$, where we may reorder the e_i. Then we may write any process in the form $\nu\vec{n}.\left(\prod e_i \middle| f\right)$ with the order of the e_i being irrelevant up to Strong Bisimulation.

Proof

Using lemma 10.13 we may reorder any processes as long as they do not include the rightmost process. For example

$$e_1|e_2|e_3|e_4 \sim^o e_2|e_1|e_3|e_4 \sim^o e_1|e_3|e_2|e_4$$

using lemma 10.13 twice. □

We may also, again upto Strong Bisimulation, push or pull the lefthand part of a parallel composition in or out under certain circumstances. We give the various cases that we will need later in the next lemma.

Lemma 10.15

For any CML_ν processes e and f and names $\vec{n}$ then:

1. For any constant function $\mathbf{c}$ then $\mathbf{c}(\nu\vec{n}.(e|f)) \sim \nu\vec{n}.(e|\mathbf{c}f)$.
2. For any recursive function g then $\nu\vec{n}.(g(e|f)) \sim \nu\vec{n}.(e|gf)$.
3. For any CML_ν process g then $\nu\vec{n}.(g|e,f) \sim \nu\vec{n}.(g|(e,f))$.
4. For CML_ν processes g_t and g_f then
   $\nu\vec{n}.(\textbf{if } e|f \textbf{ then } g_t \textbf{ else } g_f) \sim \nu\vec{n}.(e|\textbf{if } f \textbf{ then } g_t \textbf{ else } g_f)$.
5. For variable x and CML_ν process g then
   $\nu\vec{n}.(\textbf{let } x=e|f \textbf{ in } g) \sim \nu\vec{n}.(e|\textbf{let } x=f \textbf{ in } g)$.

Proof

Each case follows in essentially the same way. We give the proof of the first part as an example. We let

$$\mathcal{R} = \{(e,f) \in \text{CML}_\nu \times \text{CML}_\nu \mid e \equiv \nu\vec{n}.(e_1|\mathbf{c}(e_2|e_3)) \;\&\; f \equiv \nu\vec{n}.(e_1|(e_2|\mathbf{c}e_3))\}$$
$$\mathcal{R}' = \mathcal{R} \cup \{(e,f) \in \text{CML}_\nu \times \text{CML}_\nu \mid e \equiv f\}$$

For CML_ν processes e and f with $(e,f) \in \mathcal{R}'$ we know that $e \equiv \nu\vec{n}.(e_1|(\mathbf{c}(e_2|e_3)))$ and $f \equiv \nu\vec{n}.(e_1|e_2|\mathbf{c}e_3))$ for some $\vec{n}$, e_1, e_2 and e_3. We case split twice. First on whether we need to show that f can emulate e

and remain in the relation, or whether e can emulate f. We will deal with the first of these cases as an example since the second follows in the same way. The second case split is on what e (or f in the second case) reduces to.

[$e \searrow \nu\vec{m}.m'!v.e'$] Hence $e \equiv \nu\vec{n}.(e_1|(\mathbf{c}(e_2|e_3))) \searrow \nu\vec{m}.m'!v.e'$. By lemma 9.15 there are e'' and $\vec{m}''$ with $e_1|(\mathbf{c}(e_2|e_3)) \searrow \nu\vec{m}''.m'!v.e''$, $\vec{m}'' = \vec{m} \setminus \vec{n}$ and $e' \equiv \nu\vec{m}'''.e''$, where $\vec{m}''' = \vec{n} \setminus \vec{m}$. Then by lemma 9.16 there is a g with either $e_1 \searrow \nu\vec{m}''.m!v.g$ and $\nu\vec{m}''.g|\mathbf{c}(e_2|e_3) \equiv \nu\vec{m}''.e''$ or $\mathbf{c}(e_2|e_3) \searrow \nu\vec{m}''.m!v.g$ and $e_1|\nu\vec{m}''.g \equiv \nu\vec{m}''.e''$. We case split again.

[$e_1 \searrow \nu\vec{m}''.m'!v.g$] Therefore $e_1|e_2|\mathbf{c}e_3 \searrow \nu\vec{m}''.m'!v.(g|e_2|\mathbf{c}e_3)$, and so $\nu\vec{n}.e_1|e_2|\mathbf{c}e_3 \searrow \nu\vec{m}.m'!v.\nu\vec{m}'''.(g|e_2|\mathbf{c}e_3)$. But $e' \equiv \nu\vec{m}'''.(g|\mathbf{c}(e_2|e_3))$. Hence $f \searrow \nu\vec{m}.m'!v.f'$, with $f' = \nu\vec{m}'''.(g|e_2|\mathbf{c}e_3)$. Then for any abstraction A, $(A \bullet \nu\vec{m}' < v, e' >, A \bullet \nu\vec{m}' < v, f' >) \in \mathcal{R}$.

[$\mathbf{c}(e_2|e_3) \searrow \nu\vec{m}''.m'!v.g$] By lemma 9.8 there are e_2', e_3', v' and g' with $e_2 \equiv_\alpha e_2'$, $e_3 \equiv_\alpha e_3'$, $v \equiv v'$, $g \equiv g'$ and $\mathbf{c}(e_2'|e_3') \searrow \nu\vec{m}''.m'!v'.g'$. The last rule used in the (simple) derivation must have been CON_α. Therefore g' must have been $\mathbf{c}g''$ for some g'', and $e_2'|e_3' \searrow \nu\vec{m}''.m'!v'.g''$. Then by lemma 9.16 (again) there is an h with either $e_2' \searrow \nu\vec{m}''.m'!v'.h$ and $\nu\vec{m}''.h|e_3' \equiv \nu\vec{m}''.g''$ or $e_3' \searrow \nu\vec{m}''.m'!v'.h$ and $e_3'|\nu\vec{m}''.h \equiv \nu\vec{m}''.g''$. We case split yet again!

[$e_2' \searrow \nu\vec{m}''.m'!v'.h$] Then using PAR_2 we derive that $e_2'|\mathbf{c}e_3' \searrow \nu\vec{m}''.m'!v'.(h|\mathbf{c}e_3')$. Then by lemma 9.9 and STRUC we see that $e_2|\mathbf{c}e_3 \searrow \nu\vec{m}''.m'!v.(h|\mathbf{c}e_3)$. Then by PAR_1 we obtain $e_1|e_2|\mathbf{c}e_3 \searrow \nu\vec{m}''.m'!v.(e_1|h|\mathbf{c}e_3)$. Finally using RES rules we observe that $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \nu\vec{m}.m'!v.\nu\vec{m}'''.(e_1|h|\mathbf{c}e_3)$. Again we observe that $e' \equiv \nu\vec{m}'''.(e_1|\mathbf{c}(h|e_3))$. So $f \searrow \nu\vec{m}.m'!v.f'$ where $f' = \nu\vec{m}'''.(e_1|h|\mathbf{c}e_3)$. So for any abstraction A, $(A \bullet \nu\vec{m}' < v, e' >, A \bullet \nu\vec{m}' < v, f' >) \in \mathcal{R}$.

[$e_3 \searrow \nu\vec{m}''.m'!v'.h$] Then using CON_α and PAR_1 we derive that $e_2'|\mathbf{c}e_3' \searrow \nu\vec{m}''.m'!v'.(e_2'|h)$. Then by lemma 9.9 and STRUC we see that $e_2|\mathbf{c}e_3 \searrow \nu\vec{m}''.m'!v.(e_2|h)$. Then by PAR_1 we obtain $e_1|e_2|\mathbf{c}e_3 \searrow \nu\vec{m}''.m'!v.(e_1|e_2|h)$. Finally using RES rules we observe that $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \nu\vec{m}.m'!v.\nu\vec{m}'''.(e_1|e_2|h)$. Again

we observe that $e' \equiv \nu\vec{m}'''.(e_1|\mathbf{c}(e_2|h))$. So $f \searrow \nu\vec{m}.m'!v.f'$ where $f' = \nu\vec{m}'''.(e_1|e_2|h)$. So for any abstraction A, $(A \bullet \nu\vec{m}' < v, e' >, A \bullet \nu\vec{m}' < v, f' >) \in \mathcal{R}$.

[$e \searrow \nu\vec{m}.\surd v.e'$] This cannot occur.

[$e \searrow n?x.e'$] This follows in the same way as the $e \searrow \nu\vec{m}.m'!v.e'$ case, with the change of abstraction to concretion and the lack of the need to use lemma 9.9.

[$e \searrow \tau.e'$] Hence $e \equiv \nu\vec{n}.(e_1|(\mathbf{c}(e_2|e_3))) \searrow \tau.e'$. By lemma 9.15 there is an e'' with $e' \equiv \nu\vec{m}.e''$ and $e_1|(\mathbf{c}(e_2|e_3)) \searrow \tau.e''$. We then use lemma 9.17 and see that one of the following cases must occur:

- [$e_1 \searrow \tau.e_1'$ & $e_1'|\mathbf{c}(e_2|e_3) \equiv e''$] for some e_1'. Then, using PAR_2, $e_1|e_2|\mathbf{c}e_3 \searrow \tau.(e_1'|e_2|\mathbf{c}e_3$. Then, possibly multiple uses of, RES_τ gives $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \tau.\nu\vec{n}.(e_1'|e_2|\mathbf{c}e_3)$. But $e' \equiv \nu\vec{n}.(e_1'|\mathbf{c}(e_2|e_3))$. So $f \searrow \tau.f'$ where $f' = \nu\vec{n}.(e_1'|e_2|\mathbf{c}e_3)$ and $(e', f') \in \mathcal{R}$.
- [$\mathbf{c}(e_2|e_3) \searrow \tau.g$ & $e_1|g \equiv e''$] for some g. By lemma 9.8 there are e_2', e_3' and g' with $e_2' \equiv_\alpha e_2$, $e_3' \equiv_\alpha e_3$, $g' \equiv g$ and $\mathbf{c}(e_2'|e_3') \searrow \tau.g'$ simple. The last rule used in (simple) derivation must have been either CON_α or $\text{CON}_\surd$. We case split again!
  - **CON_α** Hence $e_2'|e_3' \searrow \tau.g''$ with $g' = \mathbf{c}g''$. The previous rule used must have been one of PAR_1, PAR_2, COM_1 or COM_2. We case split one more time.
    - **PAR_1** Hence there is an e_3'' with $e_3' \searrow \tau.e_3''$ and $g'' = e_2'|e_3''$. Therefore, using CON_α, PAR_1 twice, STRUC and RES rules, we get $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \tau.\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3'')$. But $e' \equiv \nu\vec{n}.(e_1|\mathbf{c}(e_2|e_3''))$. Therefore $f \searrow \tau.f'$ where $f' = \nu\vec{n}.(e_1|e_2|\mathbf{c}e_3'')$ and $(e', f') \in \mathcal{R}$.
    - **PAR_2** Hence there is an e_2'' with $e_2' \searrow \tau.e_2''$ and $g'' = e_2''|e_3'$. Therefore, using PAR_2, PAR_1, STRUC and RES rules, we get $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \tau.\nu\vec{n}.(e_1|e_2''|\mathbf{c}e_3)$. But $e' \equiv \nu\vec{n}.(e_1|\mathbf{c}(e_2''|e_3))$. Therefore letting $f' = \nu\vec{n}.(e_1|e_2''|\mathbf{c}e_3)$ gives $f \searrow \tau.f'$ and $(e', f') \in \mathcal{R}$.

COM$_1$ Hence there are e_2'', e_3'', $\vec{n}'$, n'' and v such that $e_2' \searrow \nu\vec{n}'.n''!v.e_2''$, $e_3' \searrow n''?x.e_3''$ and $\nu\vec{n}'.(e_2''|e_3''[v/x]) \equiv g''$. Then CON$_\alpha$, CON$_1$, PAR$_1$, STRUC and RES rules give $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \tau.\nu\vec{n}.(e_1|\nu\vec{n}'.(e_2''|\mathbf{c}(e_3''[v/x])))$. We now let $f' = \nu\vec{n}.(e_1|\nu\vec{n}'.(e_2''|\mathbf{c}(e_3''[v/x])))$. Therefore $f \searrow \tau.f'$. We also observe that $e' \equiv \nu\vec{n}.(e_1|\nu\vec{n}'.\mathbf{c}(e_2''|e_3''[v/x]))$ and therefore that $(e', f') \in \mathcal{R}$.

COM$_2$ Hence there are e_2'', e_3'', $\vec{n}'$, n'' and v such that $e_2' \searrow n''?x.e_2''$, $e_3' \searrow \nu\vec{n}'.n''!v.e_3''$ and $\nu\vec{n}'.(e_2''[v/x]|e_3'') \equiv g''$. Then CON$_\alpha$, CON$_1$, PAR$_1$, STRUC and RES rules give $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \tau.\nu\vec{n}.(e_1|\nu\vec{n}'.(e_2''[v/x]|\mathbf{c}e_3''))$. We now let $f' = \nu\vec{n}.(e_1|\nu\vec{n}'.(e_2''[v/x]|\mathbf{c}e_3''))$. Therefore $f \searrow \tau.f'$. We also observe that $e' \equiv \nu\vec{n}.(e_1|\nu\vec{n}'.\mathbf{c}(e_2''[v/x]|e_3''))$ and therefore that $(e', f') \in \mathcal{R}$.

CON$_\surd$ Hence $e_2'|e_3' \searrow \nu\vec{n}'.\surd v.g''$, with $g' = \nu\vec{n}'.(\delta(\mathbf{c}, v)|g'')$. The previous rule used in derivation must have been PAR$_1$ and so $g'' = e_2'|e_3''$ for some e_3'' with $e_3' \searrow \nu\vec{n}'.\surd v.e_3''$. Therefore $\mathbf{c}e_3' \searrow \tau.\nu\vec{n}'.(e_3''|\delta(\mathbf{c}, v))$, and PAR$_1$ gives $e_2'|\mathbf{c}e_3' \searrow \tau.e_2'|\nu\vec{n}'.(e_3''|\delta(\mathbf{c}, v))$. We use STRUC to derive $e_2|\mathbf{c}e_3 \searrow \tau.\nu\vec{n}'.(e_2|e_3''|\delta(\mathbf{c}, v))$. So using PAR$_1$ again we get $e_1|e_2|\mathbf{c}e_3 \searrow \tau.e_1|\nu\vec{n}'.(e_2|e_3''|\delta(\mathbf{c}, v))$. Then using RES rules we derive $\nu\vec{n}.(e_1|e_2|\mathbf{c}e_3) \searrow \tau.\nu\vec{n}.(e_1|\nu\vec{n}'.(e_2|e_3''|\delta(\mathbf{c}, v)))$. But we observe that $e' \equiv \nu\vec{n}.(e_1|\nu\vec{n}'.(e_2|e_3''|\delta(\mathbf{c}, v)))$. Therefore by setting $f' = \nu\vec{n}.(e_1|\nu\vec{n}'.(e_2|e_3''|\delta(\mathbf{c}, v)))$, we get $f \searrow \tau.f'$ and $e' \equiv f'$. Therefore $(e', f') \in \mathcal{R}'$.

$[e_1 \searrow \nu\vec{m}.m'!v.e_1' \ \& \ \mathbf{c}(e_2|e_3) \searrow m'?x.e''' \ \& \ \nu\vec{m}.(e_1'|e''[v/x]) \equiv e'']$ for some v, m', e_1', e''' and $\vec{m} \subseteq \mathrm{fn}(v)$. This now follows in the same way as the $e \searrow n?x.e'$ case above, starting part way through.

$[e_1 \searrow m'?x.e_1' \ \& \ \mathbf{c}(e_2|e_3) \searrow \nu\vec{m}.m'!v.e''' \ \& \ \nu\vec{m}.(e_1'[v/x]|e'') \equiv e'']$ for some v, m', e_1', e''' and $\vec{m} \subseteq \mathrm{fn}(v)$. Similarly this follows in the same way as the $e \searrow \nu\vec{m}.m'!v.e'$ case above.

□

Definition 10.16

The set of channel names contains a (countable) subset $s_0, s_1, s_2, \ldots$. We define an embedding $-^e$, on expressions and names, by rewriting all names in $-$ according to the following rules:

$$\begin{array}{lcll} s_i & \mapsto & s_{i+1} & \\ n & \mapsto & n & n \neq s_i \text{ for any } i \end{array}$$

Similarly we define $-^u$ to be the inverse of $-^e$, with the restriction that s_0 may not appear in $-$.

The main use of definition 10.16 is that we may ensure that there is one, or more, specific name that cannot occur in an embedded expression. We note that if $e \searrow \sqrt{C}$ then $e^e \searrow \sqrt{C^e}$ (and similarly for output, input or silent transitions) by observing that we can rewrite the derivation tree by the literal replacement of s_i with s_{i+1} everywhere.[4] We also note that if $e \sim^o f$ then $e^e \sim^o f^e$.[5]

We use this to prove the following "surprising" lemma which also gives the useful results of lemma 10.18.

Lemma 10.17

For any CML$_\nu$ expressions e and f with $e \sim^o f$ then for any CML$_\nu$ expression g we may deduce that $e|g \sim^o f|g$. Also for any values v and v' and names $\vec{n}$ and $\vec{m}$ then $\forall A \quad A \bullet \nu\vec{n} < v, e > \sim^o A \bullet \nu\vec{m} < v', f >$ implies that for any CML$_\nu$ expression h with x as a free variable we may deduce $\nu\vec{n}.(e|h[v/x]) \sim^o \nu\vec{m}.(f|h[v'/x])$.

Proof

We will prove the first part as a special case of the second part. We let

$$\begin{array}{l} \mathcal{R} = \Big\{ \Big(\nu\vec{n}.(e|h'), \nu\vec{n}'.(f|h'') \Big) \text{ s.t.} \\ \quad \nu\vec{n}^e.((\textbf{let } y{=}h' \textbf{ in } s_0!y)|e^e) \sim \nu\vec{n}'^e.((\textbf{let } y{=}h'' \textbf{ in } s_0!y)|f^e) \Big\} \end{array}$$

We will show that $\mathcal{R} \subseteq \{\mathcal{R}\}_r$. There are eight cases given by definition 10.8. We will only give one case since the others all follow in a similar way, as a special case of the one given. We consider the case where $\nu\vec{n}.(e|h_1) \searrow \tau.e'$. Therefore by lemma 9.15 there is an e'' with $e|h_1 \searrow \tau.e''$ and $\nu\vec{n}.e'' \equiv e'$. Therefore by lemma 9.17 there are four possible cases:

[4] Similarly for $\Downarrow$.

[5] Similarly for $\simeq^o$.

$[h_1 \searrow \tau.h_3]$ and $e'' \equiv e|h_3$. Therefore, using LET_α, PAR_2 and RES_τ many times[6], gives let $y = h_1^e$ in $s_0!y|e^e \searrow \tau.$let $y = h_3^e$ in $s_0!y|e$. Therefore $\nu\vec{n}'^e.((\text{let } y = h_2^e \text{ in } s_0!y)|f^e) \searrow \tau.f'$ and let $y = h_3^e$ in $s_0!y|e \sim f'$. Therefore by lemma 9.15 we see that there is an f'' with $\nu\vec{n}'^e.f'' \equiv f'$ and $(\text{let } y{=}h_3^e \text{ in } s_0!y)|f^e \searrow \tau.f''$. Then by lemma 9.17 there are four cases:

- $[\text{let } y{=}h_2^e \text{ in } s_0!y \searrow \tau.h_4]$ and $h_4|f^e \equiv f''$. By lemma 9.8 there are g and g' with let $y = h_2^e$ in $s_0!y \equiv_\alpha$ let $y = g$ in $s_0!y$, $g' \equiv h_4$ and let $y{=}g$ in $s_0!y \searrow \tau.g'$ simple. The last rule used in the derivation must have been LET_α and so there is a g'' with $g \searrow \tau.g''$ and $g' =$ let $y{=}g''$ in $s_0!y$. But $g \equiv_\alpha h_2^e$. Therefore $h_2 \searrow \tau.g''^u$. So $\nu\vec{n}'.(f|h_2) \searrow \tau.\nu\vec{n}'.(f|g''^u)$. We can see that, as required, $\nu\vec{n}^e.((\text{let } y{=}h_3^e \text{ in } s_0!y)|e^e) \sim \nu\vec{n}'^e.((\text{let } y{=}h_4^{ue} \text{ in } s_0!y)|f^e)$.
- $[f^e \searrow \tau.f'''']$ and $(\text{let } y = h_2^e \text{ in } s_0!y)|f'''' \equiv f''$. Therefore we deduce that $\nu\vec{n}'.(f|h_2) \searrow \nu\vec{n}'.(f''''^u|h_2)$. We then also see that, as required $\nu\vec{n}.^e.((\text{let } y{=}h_3^e \text{ in } s_0!y)|e^e) \sim \nu\vec{n}'^e.((\text{let } y{=}h_2^e \text{ in } s_0!y)|(f''''^u)^e)$.
- $[f^e \searrow m?z.f''']$ let $y{=}h_2^e$ in $s_0!y \searrow \nu\vec{m}'.m!v''.h_3$ and $\nu\vec{m}'.(h_3|f'''[v''/z])$, for some m, $\vec{m}'$ and v''. Using the same logic as above we see that $h_3 \equiv$ let $y = h_4$ in $s_0!y$ and $h_2^e \searrow \nu\vec{m}'.m!v''.h_4$. The result now follows easily in the same way as the $[\text{let } y = h_2^e \text{ in } s_0!y \searrow \tau.h_4]$ case above.
- $[\text{let } y{=}h_2^e \text{ in } s_0!y \searrow m?z.h_3]$ $f^e \searrow \nu\vec{m}'.m!v''.f'''$ and $\nu\vec{m}'.(h_3[v''/z]|f''')$, for some m, $\vec{m}'$ and v''. This follows in the same way as the previous case.

Other cases The other three cases follow in the same way. In each instance we derive that let $y = h_1^e$ in $s_0!y|e^e \searrow \tau.$let $y = h_3^e$ in $s_0!y|e$. The result then follows as in the $[h_1 \searrow \tau.h_3]$ case.

The only other case (and it's symmetrical case) we need to consider is when $\nu\vec{n}.(e|h_1) \searrow \nu\vec{m}.\surd v''.e'$. Using the same logic as above we may deduce that there are $\vec{m}'$ and h_3 with $e' \equiv \nu\vec{m}'.(e^e|h_3)$, $\vec{m}' = \vec{n} \setminus \vec{m}$ and $\nu\vec{n}.((\text{let } y{=}h_1^e \text{ in } s_o!y)|e^e) \searrow \tau.\nu\vec{m}.s_0!v''.\nu\vec{m}'.(h_3|()|e^e)$. Hence we may derive

[6] And the embedding $-^e$

that $\nu\vec{n}.((\text{let } y=h_2^e \text{ in } s_o!y)|f^e) \searrow \tau.\nu\vec{m}''.s_0!v''.\nu\vec{m}'''.(h_4|()|f^e)$ and for any abstraction A then $A \bullet \nu\vec{m} < v'', \nu\vec{m}'.(h_3|e^e) > \sim A \bullet \nu\vec{m}'' < v''', \nu\vec{m}'''.(h_4|f^e) >$[7], for some v''' and $\vec{m}''$ with $\vec{m}''' = \vec{n} \setminus \vec{m}''$. We then see that $\nu\vec{n}'.(f|h_2) \searrow \nu\vec{m}''.\surd v'''.\nu\vec{m}'''.(f|h_4^u|\Lambda)$. Recalling lemma 9.26 on page 135 we see that $e' \equiv e'|\Lambda$. Therefore by using corollary 10.14 twice we see that for any abstraction A then $A \bullet \nu\vec{m} < v'', e' > \sim A \bullet \nu\vec{m}'' < v''', \nu\vec{m}'''.(f|h_4^u|\Lambda) >$.

Therefore $\mathcal{R} \subseteq \sim_r$ and hence by corollary 10.11 $\mathcal{R} \subseteq \sim$. We extend the result to $\sim^o$ by substitution of closed expressions for free variables. The result now follows by letting the initial $h' = h[v/x]$, $h'' = h[v'/x]$ using $A = (x, \text{let } y=h \text{ in } s!y)$, where s is fresh in e, f and h. The first case follows by setting $A = (z, g)$, where $z \notin \text{fv}(g)$.

□

Lemma 10.18

For any CML_ν expressions e and f with $e \simeq^o f$ then for any $\vec{n}$, $\vec{m}$, v, v', e' and f' with $e \searrow \nu\vec{n}.\surd v.e'$, $f \searrow \nu\vec{m}.\surd v'.f'$ and for all A $A \bullet \nu\vec{n} < v, e' > \sim^o A \bullet \nu\vec{m} < v', f' >$ then:

- For any constant function c, $\nu\vec{n}.(e'|\delta(\mathbf{c}, v)) \sim^o \nu\vec{m}.(f'|\delta(\mathbf{c}, v'))$.
- For any g, $\nu\vec{n}.(e'|\text{let } x=g \text{ in } < v, x >) \sim^o \nu\vec{m}.(f'|\text{let } x=g \text{ in } < v', x >)$.
- For $v = \textbf{fix}\,(x=\textbf{fn}y \Rightarrow e_1)$ then $v' = \textbf{fix}\,(x=\textbf{fn}y \Rightarrow f_1)$ and for any CML_ν expression g, $\nu\vec{n}.(e'|\text{let } y=g \text{ in } e_1[v/x]) \sim^o \nu\vec{m}.(f'|\text{let } y=g \text{ in } f_1[v'/x])$.
- For any CML_ν expression g with x as a free variable $\nu\vec{n}.(e'|g[v/x]) \sim^o \nu\vec{m}.(f'|g[v'/x])$.

Proof

This follows directly from lemma 10.17, using various different abstractions A.

□

10.8 Properties of Evaluation Bisimulation

We now give various properties of Evaluation Bisimulation. Many of these will be used in the proof that $\simeq^o$ is a congruence relation, while others are

[7] Since $\nu\vec{n}.(e|()|f) \sim \nu\vec{n}.(e|f)$ for any $\vec{n}$, e and f.

merely useful observations. We start with two very simple lemmas that show that Evaluation Bisimulation is closed under the structural congruence and under reordering of processes within an expression with certain restrictions.

Lemma 10.19

For any CML$_\nu$ processes e_1 and e_2, $e_1 \equiv e_2$ implies that $e_1 \simeq^o e_2$.

Proof

If e_1 and e_2 have no free variables then for any CML$_\nu$ expression f, $f|e_1 \Downarrow C$ implies that $f|e_2 \Downarrow C$ using (STRUC), and $\simeq^o$ is an equivalence relation. If e_1 and e_2 do have free variables then the previous argument applies for all substitutions of values for free variables. □

Lemma 10.20

For any CML$_\nu$ expressions e, f and g then $e \simeq^o f$ implies that $g|e \simeq^o g|f$.

Proof

This follows easily from the definition 10.4, observing that we are just restricting the parallel contexts. □

Lemma 10.21

$\nu\vec{n}.(e_1|e_2|e_3) \simeq^o \nu\vec{n}.(e_2|e_1|e_3)$

Proof

This follows directly from lemmas 10.13 and 10.12. □

Corollary 10.22

Let $\prod e_1 = e_1|e_2|\ldots|e_n$, where we may reorder the e_i. Then we may write any process in the form $\nu\vec{n}.\left(\prod e_i \middle| f\right)$ with the order of the e_i being irrelevant up to Evaluation Bisimulation.

Proof

Using lemma 10.21 we may reorder any processes as long as they do not include the rightmost process. For example

$$e_1|e_2|e_3|e_4 \simeq^o e_2|e_1|e_3|e_4 \simeq^o e_1|e_3|e_2|e_4$$

using lemma 10.21 twice. □

We now show that Evaluation Bisimulation implies that if an expression can do a τ action then an equivalent expression can match it with zero or more τ actions and the continuations remain equivalent.

Lemma 10.23

For any CML$_\nu$ processes e and f, $e \simeq f$ and $e \searrow \tau^*.e'$ implies that there is an f' such that $f \searrow \tau^*.f'$ and $e' \simeq f'$.

Proof

We consider $e \searrow \tau^i.e'$. If $i = 0$ then the result follows trivially. Otherwise we take a fresh name m and $m!\mathbf{true}|e \searrow \tau^i.m!\mathbf{true}.(()|e')$. Then by lemma 9.34 $m!\mathbf{true}|e \Downarrow m!\mathbf{true}.(()|e')$. Hence, using the abstractions $(x, \mathbf{if}\ x\ \mathbf{then}\ m'!\mathbf{true}\ \mathbf{else}\ m''!\mathbf{true})$ and (x, Λ) for fresh names m' and m'', $m!\mathbf{true}|f \Downarrow m!\mathbf{true}.f'$ with $()|e' \simeq f'$. By lemma 9.12 there is an f'' with $f' \equiv ()|f''$ and $f \searrow \tau^*.f''$. By lemma 10.27 $e' \simeq ()|e'$ and $f' \simeq f''$ using lemma 10.19 as well. Therefore $e' \simeq f''$ as required. □

We now extend lemma 10.20 to show that, just as we could for Strong Bisimulation, we can define Evaluation Bisimulation without having a parallel context.

Definition 10.24 *Reduced Evaluation Bisimulation for CML$_\nu$*

Given $\mathcal{R} \subseteq \text{CML}_\nu \times \text{CML}_\nu$, define $e_1[\mathcal{R}]_r e_2$ to hold if and only if for any n, a channel name, then:

$$
\begin{array}{lllll}
 & e_1 \searrow n!C \Rightarrow \exists D\ [e_2 \Downarrow n!D & \& & \forall A\ (A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & e_1 \searrow \surd C \Rightarrow \exists D\ [e_2 \Downarrow \surd D & \& & \forall A\ (A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & e_1 \searrow n?A \Rightarrow \exists B\ [e_2 \Downarrow n?B & \& & \forall C\ (C \bullet A)\mathcal{R}(C \bullet B)] \\
\& & e_1 \searrow \tau.e_1' \Rightarrow \exists e_2'\ [e_2 \searrow \tau^*.e_2' & \& & e_1'\mathcal{R}e_2'] \\
 & & & & \\
\& & e_2 \searrow n!C \Rightarrow \exists D\ [e_1 \Downarrow n!D & \& & \forall A\ (A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & e_2 \searrow \surd C \Rightarrow \exists D\ [e_1 \Downarrow \surd D & \& & \forall A\ (A \bullet C)\mathcal{R}(A \bullet D)] \\
\& & e_2 \searrow n?A \Rightarrow \exists B\ [e_1 \Downarrow n?B & \& & \forall C\ (C \bullet A)\mathcal{R}(C \bullet B)] \\
\& & e_2 \searrow \tau.e_2' \Rightarrow \exists e_1'\ [e_1 \searrow \tau^*.e_1' & \& & e_1'\mathcal{R}e_2']
\end{array}
$$

Then we let $\simeq_r$ be $\nu\mathcal{R}.[\mathcal{R}]_r$. We let $\simeq_r^o$ be the extension to open processes by substitution of closed values for free variables.

Lemma 10.25

For any CML$_\nu$ expressions e, f and g then $e \simeq^o_r f$ implies that $g|e \simeq^o_r g|f$.

Proof

This follows in the same way as lemma 10.10. □

Corollary 10.26

$\simeq^o$ and $\simeq^o_r$ cooincide.

Proof

This follows directly from definitions 10.4 and 10.24, lemmas 10.23 and 10.25 and theorem 9.29. □

Most of the rest of the results in the chapter are used in the proof of congruence. They depend heavily on the transition relation, and the lemmas in chapter 9, for their proofs.

Lemma 10.27

For any CML$_\nu$ expressions e and f, value v and names $\vec{n}$ then $\nu\vec{n}.(e|v|f) \simeq^o \nu\vec{n}.(e|f)$.

Proof

We proceed by a two step method. If $\nu\vec{n}.(e|v|f)$ has no free variables then we let $\mathcal{R} = \{(g,h)$ s.t. $\exists e, f, \vec{n}\ \ g \equiv \nu\vec{n}.(e|f)\ \&\ h \equiv \nu\vec{n}.(e|v|f)\}$. We show that $\mathcal{R}$ is a fixed point of $[-]$ using corollary 9.21 and lemma 9.24, and hence that $\mathcal{R}$ is contained in $\simeq$. We then extend the result to $\simeq^o$ since the argument still holds for any substitutions of closed values for free variables. □

Lemma 10.28

For any CML$_\nu$ processes e and f and names $\vec{n}$ then:

1. For any constant function $\mathbf{c}$ then $\mathbf{c}(\nu\vec{n}.(e|f)) \simeq \nu\vec{n}.(e|\mathbf{c}f)$.
2. For any recursive function g then $\nu\vec{n}.(g(e|f)) \simeq \nu\vec{n}.(e|gf)$.
3. For any CML$_\nu$ process g then $\nu\vec{n}.(g|e,f) \simeq \nu\vec{n}.(g|(e,f))$.
4. For CML$_\nu$ processes g_t and g_f then $\nu\vec{n}.(\mathbf{if}\ e|f\ \mathbf{then}\ g_t\ \mathbf{else}\ g_f) \simeq \nu\vec{n}.(e|\mathbf{if}\ f\ \mathbf{then}\ g_t\ \mathbf{else}\ g_f)$.
5. For variable x and CML$_\nu$ process g then $\nu\vec{n}.(\mathbf{let}\ x{=}e|f\ \mathbf{in}\ g) \simeq \nu\vec{n}.(e|\mathbf{let}\ x{=}f\ \mathbf{in}\ g)$.

Proof

These all follow directly from lemmas 10.12 and 10.15. □

The next two lemmas are required to prove the Evaluation Bisimulation version of lemma 10.17.

Lemma 10.29

For any value v, expression e, names $\vec{n}$ and any constant function c then $\nu\vec{n}.(e|\mathbf{c}v) \simeq^o \nu\vec{n}.(e|\delta(\mathbf{c},v))$.

Proof

We let $\mathcal{R} = \{(f,g) \text{ s.t. } f \equiv \nu\vec{n}.(e|\mathbf{c}v) \quad \& \quad g \equiv \nu\vec{n}.(e|\delta(\mathbf{c},v))\}$. We then let $\mathcal{R}' = \{(f,g) \text{ s.t. } f \equiv g \text{ or } (f,g) \in \mathcal{R}\}$. For any f and g with $(f,g) \in \mathcal{R}$ then there are $e, v, \vec{n}$ and c with $f \equiv \nu\vec{n}.(e|\mathbf{c}v)$ and $g \equiv \nu\vec{n}.(e|\delta(\mathbf{c},v))$. For any names $\vec{m}$, label l and expression g', then $g \searrow \nu\vec{m}.l.g'$ implies that $f \searrow \tau.\nu\vec{m}.l.g'$ and $(g',g') \in \mathcal{R}'$. Therefore we only need to consider what transitions f may make. We case split.

[$f \searrow m?x.f'$] By lemma 9.15, remembering that $f \equiv \nu\vec{n}.(e|\mathbf{c}v)$, there is an f'' with $e|\mathbf{c}v \searrow m?x.f''$ and $f' \equiv \nu\vec{n}.f''$. Then by lemma 9.16 we see that there is an e' with $e \searrow m?x.e'$ and $f'' \equiv e'|\mathbf{c}v$ (since $\mathbf{c}v$ cannot input). Therefore $\nu\vec{n}.(e|\delta(\mathbf{c},v)) \searrow m?x.\nu\vec{n}.(e'|\delta(\mathbf{c},v))$ and for any concretion C then $(C \bullet (x,f'), C \bullet (x, \nu\vec{n}.(e'|\delta(\mathbf{c},v)))) \in \mathcal{R}$.

[$f \searrow \nu\vec{m}'.m!v'.f'$] By lemma 9.15, remembering that $f \equiv \nu\vec{n}.(e|\mathbf{c}v)$, there are f'', $\vec{n}'$ and $\vec{n}''$ with $e|\mathbf{c}v \searrow \nu\vec{n}'.m!v.f''$ and $f' \equiv \nu\vec{n}''.f''$, where $\vec{n}' = \vec{m}' \setminus \vec{n}$ and $\vec{n}'' = \vec{n} \setminus \vec{m}'$. Then by lemma 9.16 we see that there is an e' with $e \searrow \nu\vec{n}'.m!v.e'$ and $f'' \equiv e'|\mathbf{c}v$ (since $\mathbf{c}v$ cannot output). Therefore $\nu\vec{n}.(e|\delta(\mathbf{c},v)) \searrow \nu\vec{n}'.m!v.\nu\vec{n}''.(e'|\delta(\mathbf{c},v))$ and for any abstraction A then $(A \bullet \nu\vec{m} < v, f' >, A \bullet \nu\vec{n}' < v, \nu\vec{n}''.(e'|\delta(\mathbf{c},v)) >) \in \mathcal{R}$.

[$f \searrow \nu\vec{m}.\sqrt{v'}.f'$] Cannot occur.

[$f \searrow \tau.f'$] By lemma 9.15 there is an f'' with $e|\mathbf{c}v \searrow \tau.f''$ and $f' \equiv \nu\vec{n}.f''$. Then by lemma 9.17 either $e \searrow \tau.e'$, for some e' and $f'' \equiv e'|\mathbf{c}v$ or $\mathbf{c}v \searrow \tau.\Lambda|\delta(\mathbf{c},v)$ and $f'' \equiv e|\delta(\mathbf{c},v)$. In the latter case we have $f' \equiv g$ and so $(f',g) \in \mathcal{R}'$. In the former case, using PAR_2 and RES_τ, we derive that $\nu\vec{n}.(e|\delta(\mathbf{c},v)) \searrow \tau.\nu\vec{n}.(e'|\delta(\mathbf{c},v))$ and $(\nu\vec{n}.(e'|\mathbf{c}v), \nu\vec{n}.(e'|\delta(\mathbf{c},v))) \in \mathcal{R}$ as required.

Therefore $\mathcal{R}' \subseteq \simeq^o_r$ and hence the result follows by corollary 10.26. □

Lemma 10.30

For any CML$_\nu$ expression e, value v and names $\vec{n}$ and m then $\nu\vec{n}.(\textbf{let } x=v \textbf{ in } m!x|e) \simeq^o \nu\vec{n}.(m!v|e)$.

Proof

The proof follows in the same way as lemma 10.29, except that the case where a value is returned can occur. However this follows in just the same way as the output case above. □

Lemma 10.31

For any CML$_\nu$ expressions e and f with $e \simeq^o f$ then for any CML$_\nu$ expression g we may deduce that $e|g \simeq^o f|g$. Also for any values v and v' and names $\vec{n}$ and $\vec{m}$ then $\forall A \quad A \bullet \nu\vec{n} < v, e > \simeq^o A \bullet \vec{m} < v', f >$ implies that for any CML$_\nu$ expression h with x as a free variable we may deduce $\nu\vec{n}.(e|h[v/x]) \simeq^o \nu\vec{m}.(f|g[v'/x])$.

Proof

This follows in the same way as lemma 10.17 with the addition of the use of corollary 9.35 and lemma 10.30. □

Lemma 10.32

For any CML$_\nu$ expressions e, e', f and f', names $\vec{n}$ and $\vec{m}$ and values v and v' with $e \simeq^o f$, $e \Downarrow \nu\vec{n}.\surd v.e'$, $f \Downarrow \nu\vec{m}.\surd v'.f'$ and for all A, $A \bullet \nu\vec{n} < v, e' > \simeq^o A \bullet \nu\vec{m} < v', f' >$ then:

- For any constant function $\mathbf{c}$, $\nu\vec{n}.(e'|\delta(\mathbf{c}, v)) \simeq^o \nu\vec{m}.(f'|\delta(\mathbf{c}, v'))$.
- For any g, $\nu\vec{n}.(e'|\textbf{let } x=g \textbf{ in } < v, x >) \simeq^o \nu\vec{m}.(f'|\textbf{let } x=g \textbf{ in } < v', x >)$.
- For $v = \textbf{fix}\,(x=\textbf{fn}y \Rightarrow e_1)$ then $v' = \textbf{fix}\,(x=\textbf{fn}y \Rightarrow f_1)$ and for any CML$_\nu$ expression g, $\nu\vec{n}.(e'|\textbf{let } y=g \textbf{ in } e_1[v/x]) \simeq^o \nu\vec{m}.(f'|\textbf{let } y=g \textbf{ in } f_1[v'/x])$.
- For any CML$_\nu$ expression g with x as a free variable $\nu\vec{n}.(e'|g[v/x]) \simeq^o \nu\vec{m}.(f'|g[v'/x])$.

Proof

This follows directly from lemma 10.31, using various different abstractions A.

□

10.9 Some basic examples of equivalent expressions

In this section we give some examples of equivalent expressions. We first start by examining, three monad laws from Moggi's [Mog91] monadic metalanguage, and showing that they are Evaluation bisimilar. These are the left unit, the associativity and the right unit equations.

Lemma 10.33

For CML$_\nu$ expressions e, f and g, value v and variables x and y then

1. $\mathbf{let}\ x{=}v\ \mathbf{in}\ e \simeq e[v/x]$
2. $\mathbf{let}\ y{=}(\mathbf{let}\ x{=}e\ \mathbf{in}\ f)\ \mathbf{in}\ g \simeq \mathbf{let}\ x{=}e\ \mathbf{in}\ (\mathbf{let}\ y{=}f\ \mathbf{in}\ g)$
3. $\mathbf{let}\ x{=}e\ \mathbf{in}\ x \simeq e$

Proof

For the first part we let $\mathcal{R} = \{(\mathbf{let}\ x = v\ \mathbf{in}\ e, e[v/x])\} \cup \{(f,g)\ \text{s.t.}\ f \equiv g\}$. We then show that $\mathcal{R}$ is closed under $[-]_r$. $\mathbf{let}\ x = v\ \mathbf{in}\ e$ only has one possible transition and that is to $e[v/x]$[8]. $e[v/x]$ can match this by doing nothing and $(e[v/x], e[v/x])$ is in $\mathcal{R}$. Furthermore two structurally equivalent expressions can always continue to mimic each other. In the other direction, if $e[v/x] \searrow n!C$ for some n and C then $\mathbf{let}\ x = v\ \mathbf{in}\ e \searrow \tau.n!C$. Similarly for $e[v/x] \searrow n?A$, $e[v/x] \searrow \surd C$ and $e[v/x] \searrow \tau.e'$. In each case we end up with identical expressions.

For the second part we prove the stronger result that

$$\mathbf{let}\ y{=}(\mathbf{let}\ x{=}e\ \mathbf{in}\ f)\ \mathbf{in}\ g \sim \mathbf{let}\ x{=}e\ \mathbf{in}\ (\mathbf{let}\ y{=}f\ \mathbf{in}\ g)$$

This result follows in exactly the same way as the proof of lemma 10.15. To prove the stated second part we then use lemma 10.12.

The third part follows by using the same method as is used in the proof of lemma 10.29. □

We next go on to show that we may introduce a "new" control operator to the language together with it's own evaluation and transition rules. This is the ! operator that is used in various languages, for example in the π-calculus and the associate language Pict. Essentially $!e$ is the same as having as many

[8] Or any structurally equivalent expression.

copies of e as needed. We will show that this is in fact just an encoding of a particular recursive function. The evaluation rule for ! is:

$$EXP \quad \frac{\mathcal{E}[e|!e] \Downarrow \nu\vec{n}.l.e'}{\mathcal{E}[!e] \Downarrow \nu\vec{n}.l.(f|e')}$$

while the transition rule is:

$$EXP \quad \frac{}{!e \searrow \tau.(e|!e)}$$

That the two rules are equivalent to each other follows from lemma 9.30 and theorem 9.29. In particular we claim that

$$\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,() \simeq\, !e$$

However, we first need to show some intermediate results.

Lemma 10.34

For any CML$_\nu$ expression e and variables x and y then

1. $\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,() \simeq \mathbf{let}\ y=()\ \mathbf{in}\ (e|\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,())$
2. $\mathbf{let}\ y=()\ \mathbf{in}\ (e|\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,()) \simeq e|\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,()$
3. $\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,() \simeq e|\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,()$

Proof

The first two parts follow in the same way as the first part of lemma 10.33. The third part follows from the first two. □

We can now also see that

$$\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,() \searrow \tau^2.(e|\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,())$$

This leads us to the following lemma.

Lemma 10.35

For any CML$_\nu$ expression e then $!e \simeq \mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,()$.

Proof

We let $\mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2$, where

$$\mathcal{R}_1 = \{(e_1,e_2)\ \text{s.t.}\ e_1 \equiv f|!e\ \&\ e_2 \equiv f|\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,()\}$$
$$\mathcal{R}_2 = \{(e_1,e_2)\ \text{s.t.}\ e_1 \equiv f|!e\ \&\ e_2 \equiv f|\mathbf{let}\ y=()\ \mathbf{in}\ (e|\mathbf{fix}\,(x=\mathbf{fn}y\Rightarrow(e|x()))\,())\}$$

If $(e_1, e_2) \in \mathcal{R}$ then $e_1 \equiv f|!e$ for some f and e_2 is structurally equivalent either $f|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,()$ or $f|\mathbf{let}\,y = ()\,\mathbf{in}\,(e|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,())$. We only consider the case where $e_2 \equiv f|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,()$ since the other case follows in essentially the same way. If $e_1 \searrow \nu\vec{n}.l.e_1'$ then we case split on whether $l = \tau$ or not. If $l = \tau$ then by lemma 9.17 there are two possibilities[9] and these are that $f \searrow \tau.f'$ and $e_1' \equiv f'|!e$ or $!e \searrow \tau.e|!e$ and $e_1' \equiv f|e|!e$. These cases are matched by

$$f|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,() \searrow \tau.f'|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,()$$
$$f|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,() \searrow \tau^2.f|e|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,()$$

If $l \neq \tau$ then by lemma 9.16 there is a f' with $f \searrow \nu\vec{n}.l.f'$ and $e_1' \equiv \nu\vec{n}.l.f'|!e$. This is then matched by

$$f|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,()) \searrow \nu\vec{n}.l.f'|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,())$$

and for any abstraction or concretion the results will still be in $\mathcal{R}$.

If $e_2 \searrow \nu\vec{n}.l.e_2'$ then we again case split depeding on whether $l = \tau$ or not. If $l = \tau$ then there are two possibilities given by lemma 9.17. These are that $f \searrow \tau.f'$ and $e_2' \equiv f'|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,()$ or that $\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,() \searrow \tau.\mathbf{let}\,y{=}()\,\mathbf{in}\,(e|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,())$ and $e_2' \equiv f|\mathbf{let}\,y{=}()\,\mathbf{in}\,(e|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,())$. The first of these cases is matched by $e_1 \searrow \tau.f'|!e$ and the second is matched by e_1 doing nothing[10]. If $l \neq \tau$ then by lemma 9.16 then there is a f' with $f \searrow \nu\vec{n}.l.f'$ and $e_2' \equiv f'|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,()$. Then this is matched by $e_1 \searrow \nu\vec{n}.l.(f'|!e)$ and for any concretion of abstraction, as appropriate, the pair will remain in $\mathcal{R}$.

Therefore $\mathcal{R} \subseteq [\mathcal{R}]_r$ and so $\mathcal{R} \subseteq \simeq$ by corollary 10.26.

□

We now turn to one final example. We define two expressions:

$$e = ((n![\Lambda]) \Rightarrow (\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(n!\mathbf{true}|x()))))|\Lambda$$
$$f = ((n![n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)]) \Rightarrow (\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(n!\mathbf{true}|x()))))|\Lambda$$

We will show that these two expressions are evaluation bisimilar. This may be rather surprising since e can transmit a delayed expression that

[9] The two other possibilities of communication between f and $!e$ cannot occur here.

[10] in the $e_2 \equiv f|\mathbf{let}\,y{=}()\,\mathbf{in}\,(e|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(e|x()))\,())$ it is matched by $e_1 \searrow \tau.f|e|!e$.

cannot do anything, whereas f can transmit a delayed expression which can transmit on channel n. However we can never examine the transmitted value in isolation, and the context can always transmit on n at any time[11] once the delayed expression has been transmitted on channel n.

Lemma 10.36

For CML$_\nu$ expressions e and f defined above then $e \simeq f$.

Proof

We let $\mathcal{R}$ be

$$\begin{aligned}\mathcal{R} = \{(e,f)\} \cup \{(e',f') \text{ s.t. } \exists h \quad & [e' \equiv h[\vec{e}/\vec{x}]|e''\Lambda \\ & \& \ f' \equiv h[\vec{f}/\vec{x}]|f''|\Lambda \\ & \& \ e'' \simeq \mathbf{fix}\,(x{=}\mathbf{fn}y \Rightarrow (n!\mathbf{true}|x()))\,() \simeq f'']\}\end{aligned}$$

where $\vec{x} = (x_1, x_2, x_3, x_4, x_5)$ and

$$\begin{aligned} e_1 &= [\Lambda] & f_1 &= [n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)] \\ e_2 &= \Lambda & f_2 &= n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda) \\ e_3 &= \Lambda & f_3 &= \mathbf{fn}\,(x \Rightarrow \Lambda)\,() \\ e_4 &= \Lambda & f_4 &= \mathbf{let}\ x{=}()\ \mathbf{in}\ \Lambda \\ e_5 &= \Lambda & f_5 &= \Lambda \end{aligned}$$

We shall show that $\mathcal{R} \subseteq [\mathcal{R}]_r$ and hence that $\mathcal{R} \subseteq \simeq$. If $(g_1, g_2) \in \mathcal{R}$ then we case split on whether $g_1 = e$ or not. If it is then $g_2 = f$. e and f each only have one transition, which is to transmit on channel n. The transition for e is

$$e \searrow n![\Lambda].\mathbf{fix}\,(x{=}\mathbf{fn}y \Rightarrow (n!\mathbf{true}|x()))\,()|\Lambda$$

f may mimic this with

$$f \searrow n![n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda))].\mathbf{fix}\,(x{=}\mathbf{fn}y \Rightarrow (n!\mathbf{true}|x()))\,()|\Lambda$$

For any concretion C, then

$$e' = C\bullet < [\Lambda], \mathbf{fix}\,(x{=}\mathbf{fn}y \Rightarrow (n!\mathbf{true}|x()))\,()|\Lambda >$$

$$f' = C\bullet < [(n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda))], \mathbf{fix}\,(x{=}\mathbf{fn}y \Rightarrow (n!\mathbf{true}|x()))\,()|\Lambda >$$

and $(e', f') \in \mathcal{R}$, since if $C = (z, h')$ then we take $h = h'[x_1/z]$.

If $g_1 \neq e$ then there are h, e'' and f'' as defined by $\mathcal{R}$. We note that neither g_1 nor g_2 can return a value. If $g_1 \searrow \nu\vec{m}.\alpha_\tau.g_1'$ then we case split on what α_τ is:

[11] Ignoring silent actions.

[$\alpha_\tau = m'?x$] Then by lemma 9.16 we may deduce that $g_1' \equiv h''|(e''|\Lambda)$ and $h[\vec{e}/\vec{x}] \searrow m'?x.h'$, for some h'. We can now prove by induction on the depth of the derivation tree, using the same sort of case splitting logic as used in the proof of lemma 10.15, that h' is of the form $h''[\vec{e}/\vec{x}]$ and that $g_2 \searrow m'?x.h''[\vec{f}/\vec{x}]|(f''|\Lambda)$. Then for any abstraction A we can see that $(A \bullet (x, h''[\vec{e}/\vec{x}]|(e''|\Lambda)), A \bullet (x, h''[\vec{f}/\vec{x}]|(f''|\Lambda))) \in \mathcal{R}$, as required.

[$\alpha_\tau = m'!v$] We case split again depending on whether $m' = n$ or not.

[$m' \neq n$] Again by lemma 9.16 we may deduce that $g_1' \equiv h'|(e''|\Lambda)$ and $h[\vec{e}/\vec{x}] \searrow \nu\vec{m}.m'!v.h'$, for some h'. The result now follows in the same was as the $m'?x$ case above.

[$m' = n$] Using lemma 9.16 we may deduce that either $g_1' \equiv h'|(e''|\Lambda)$ and $h[\vec{e}/\vec{x}] \searrow \nu\vec{m}.n!v.h'$, for some h' or else $g_1' \equiv h[\vec{e}/\vec{x}]|(e''|\Lambda)$ and $e''|\Lambda \searrow n!\mathbf{true}.e'''|\Lambda$[12]. We deal with these two cases separately.

1. So $h[\vec{e}/\vec{x}] \searrow \nu\vec{m}.n!v.h'$. The result now follows in the same way as the $m'?x$ case above.
2. So $e''|\Lambda \searrow n!\mathbf{true}.e'''|\Lambda$. But $f''|\Lambda \searrow \tau^i.n!\mathbf{true}.f'''|\Lambda$ for some $i \leq 3$. The result now follows by deducing from lemma 10.34 that $e''' \simeq \mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow (n!\mathbf{true}|x()))\,() \simeq f'''$.

[$\alpha_\tau = \tau$] We use lemma 9.17 to show that there are four possibilities:

1. $e''|\Lambda \searrow \tau.e'''$ and $g_1' \equiv h[\vec{e}/\vec{x}]|e'''$. Then $(g_1', g_2) \in \mathcal{R}$ by lemma 10.34.
2. $h[\vec{e}/\vec{x}] \searrow \tau.h'$ and $g_1' \equiv h'|(e''|\Lambda)$. This splits into two cases. The first is where there is an evaluation context $\mathcal{E}[-]$ such that $\mathcal{E}[\mathbf{sync}[\Lambda] \searrow \tau.h''[\vec{e}/\vec{x}]$, $h''[\vec{e}/\vec{x}] \equiv h'$ and h'' is the same as h except that one x_1 has become an x_2. This may be mimicked by $h[\vec{f}/\vec{x}]|f''|\Lambda \searrow \tau.h''[\vec{f}/\vec{x}]|f''|\Lambda$.

   The other case, in which the x_i's remain the same follows in the same way as the $m'?x$ case above.
3. $h[\vec{e}/\vec{x}] \searrow \nu\vec{m}'.m''!v.h'$, $e''|\Lambda \searrow m''?x.e'''$ and $g_1' \equiv \nu\vec{m}'.(h'|e'''[v/x])$. This case cannot occur since e'' cannot ever receive an input.

[12] Using lemma 9.16 again.

4. $h[\vec{e}/\vec{x}] \searrow m'?x.h'$, $e''|\Lambda \searrow \nu\vec{m}''.m'!v.e'''$ and $g_1' \equiv \nu\vec{m}''.(h'[v/x]||e''')$. Then $\vec{m}'' = \emptyset$, $m' = n$ and $v = \mathbf{true}$. The result now follows by following the $\alpha_\tau = m'?x$ and $\alpha_\tau = m'!v$ cases above.

The symmetric case of g_1 mimicking g_2 follows in essentially the same way. We just give the differences here. If $\alpha_\tau = n!\mathbf{true}$ then we may also get the case where $\mathcal{E}[n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)]||f''|\Lambda \searrow n!\mathbf{true}.\mathcal{E}[\mathbf{fn}\,(x \Rightarrow \Lambda)\,()]||f''|\Lambda$, i.e. one x_2 becomes an x_3, and we call the new expression h'. Then if $e = h[\vec{e}/\vec{x}]|e''|\Lambda$ and $e'' = e_1''|e_2''$, where e_2'' is one of

- $\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}n!\mathbf{true}|x())$
- $\mathbf{let}\ y{=}()\ \mathbf{in}\ n!\mathbf{true}|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}n!\mathbf{true}|x())\,()$
- $n!\mathbf{true}|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}n!\mathbf{true}|x())\,()$

then, noting that $h[\vec{e}/\vec{x}] = h'[\vec{e}/\vec{x}]$, e may mimic f in the following way:

$$h[\vec{e}/\vec{x}]|e''|\Lambda \searrow \tau^*.n!\mathbf{true}.h'[\vec{e}/\vec{x}]|e_1''|()|\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}n!\mathbf{true}|x())\,|\Lambda$$

Similarly in the case where $\alpha_\tau = \tau$ we get two extra cases, both of which occur in subcase 2 above. The first is where the COM rule is used and an x_2 becomes an x_3 as immediately above. The same method of mimicking applies here as well. The second is where $\mathcal{E}[\mathbf{fn}\,(x \Rightarrow \Lambda)\,()] \searrow \tau.\mathcal{E}[\mathbf{let}\ x = ()\ \mathbf{in}\ \Lambda]$ or $\mathcal{E}[\mathbf{let}\ x{=}()\ \mathbf{in}\ \Lambda] \searrow \tau.\mathcal{E}[\Lambda]$, i.e. a x_3 becomes an x_4 or an x_4 becomes an x_5. In either case e may mimic f by doing nothing.

Therefore $[\mathcal{R}]_r \subseteq \mathcal{R}$ and so $\mathcal{R} \subseteq \simeq_r$. Hence by corollary 10.26 $\mathcal{R} \subseteq \simeq$, as required.

□

10.10 Relationship to μCML

In this section we will concentrate on the fragment of CML$_\nu$ without name restriction (and without always because of the difference in usage between CML$_\nu$ and μCML$^+$). This is the same as the fragment of μCML$^+$, given in [FHJ95], without always and A. We shall call this fragment CML$_\chi$ and also take CML$_\chi$ to be the set of closed expressions in CML$_\chi$.

In [FHJ95] it is noted that the weakest equivalence defined, $\approx^h$, is a congruence relation under the condition that always and A are not allowed. This is indeed the condition for the common fragments of CML_ν and μCML^+. We shall consider how $\approx^h$ compares to Evaluation Bisimulation on this fragment. First we define $\approx^h$ in our notation.

Definition 10.37 *Higher-order Weak Bisimulation*

Given $\mathcal{R} \subseteq \text{CML}_{\not\nu} \times \text{CML}_{\not\nu}$, define $e_1[\mathcal{R}]_h e_2$ to hold if and only if

$$\begin{array}{lllll}
& e_1 \searrow n!l_1.e_1' \Rightarrow \exists\, l_2, e_2' & [e_2 \Downarrow n!l_2.e_2' & \& & l_1\mathcal{R}^l l_2 \;\&\; e_1'\mathcal{R}e_2'] \\
\& & e_1 \searrow \surd l_1.e_1' \Rightarrow \exists\, l_2, e_2' & [e_2 \Downarrow \surd l_2.e_2' & \& & l_1\mathcal{R}^l l_2 \;\&\; e_1'\mathcal{R}e_2'] \\
\& & e_1 \searrow n?x.e_1' \Rightarrow \exists\, e_2' & [e_2 \Downarrow n?x.e_2' & \& & \forall v\ e_1'[v/x]\mathcal{R}e_2'[v/x]] \\
\& & e_1 \searrow \tau.e_1' \Rightarrow \exists e_2' & [e_2 \searrow \tau^*.e_2' & \& & e_1'\mathcal{R}e_2']
\end{array}$$

$$\begin{array}{lllll}
\& & e_2 \searrow n!l_2.e_2' \Rightarrow \exists\, l_1, e_1' & [e_1 \Downarrow n!l_1.e_1' & \& & l_1\mathcal{R}^l l_2 \;\&\; e_1'\mathcal{R}e_2'] \\
\& & e_2 \searrow \surd l_2.e_2' \Rightarrow \exists\, l_1, e_1' & [e_1 \Downarrow \surd l_1.e_1' & \& & l_1\mathcal{R}^l l_2 \;\&\; e_1'\mathcal{R}e_2'] \\
\& & e_2 \searrow n?x.e_2' \Rightarrow \exists\, e_1' & [e_1 \Downarrow n?x.e_1' & \& & \forall v\ e_1'[v/x]\mathcal{R}e_2'[v/x]] \\
\& & e_2 \searrow \tau.e_2' \Rightarrow \exists e_1' & [e_1 \searrow \tau^*.e_1' & \& & e_1'\mathcal{R}e_2']
\end{array}$$

where $\mathcal{R}^l$ is defined by:

- if $v\mathcal{R}^l w$ and v is a literal then $v = w$.
- if $< v_1, v_2 > \mathcal{R}^l < w_1, w_2 >$ then $v_1\mathcal{R}^l w_1$ and $v_2\mathcal{R}^l w_2$.
- if $[ge_1]\mathcal{R}^l[ge_2]$ then $ge_1\mathcal{R}ge_2$.
- if $v\mathcal{R}^l w$ and v is a recursive function then for any u we have $vu\mathcal{R}wu$.

Then we let $\approx^h$ be $\nu\mathcal{R}.[\mathcal{R}]_h$.

$\approx^h$ is a congruence relation, and so if $e_1 \approx^h e_2$ and $l_1 \approx^{h^l} l_2$ then for any f in $\text{CML}_{\not\nu}$ we have $f[l_1/x]||e_1 \approx^h f[l_2/x]||e_2$. Therefore we have the following lemma:

Lemma 10.38

For any $\text{CML}_{\not\nu}$ expressions e and f then $e \approx^h f$ implies that $e \simeq f$.

Proof

This follows directly from the the definitions 10.24 and 10.37 and corollary 10.26, remembering what pseudo-application is shorthand for. □

This means that Evaluation Bisimulation is at least as weak as any of the equivalences given in [FHJ95], which suggests that we have at least satisfied point (5) of our desires for our bisimulation (see page 145). The obvious next question is whether the two bisimulations are the same or not. The following two expressions, from the previous section, show that Evaluation Bisimulation is strictly weaker than Higher-order Weak Bisimulation:

$$e = ((n![\Lambda]) \Rightarrow (\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(n!\mathbf{true}|x()))))|\Lambda$$
$$f = ((n![n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)) \Rightarrow (\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}(n!\mathbf{true}|x()))))|\Lambda$$

Now clearly $e \not\approx^h f$ since

$$e \searrow n![\Lambda].\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}n!\mathbf{true}|x())\,()|\Lambda$$

whilst f can only match this with

$$f \searrow n![n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)].\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}n!\mathbf{true}|x())\,()|\Lambda$$

and

$$\Lambda \not\approx^{h^l} n!\mathbf{true} \Rightarrow \mathbf{fn}\,(x \Rightarrow \Lambda)$$

However we have shown in lemma 10.36 that $e \simeq f$.

Chapter 11

Auxiliary Relation for CML_{ν}

We follow the same pattern as the proof of congruence for Evaluation Bisimulation for NCCS and proceed by a modified version of Howe's method [How96]. The approach we use here is the same as in chapter 4. We do not give details here, but follow the pattern of chapter 4 together with the locations of the various parts of the proof. In this chapter we define the auxiliary relation and prove some of the required properties. In particular we define the auxiliary relation in section 11.1. We then go on to show in section 11.2 that the auxiliary relation contains Evaluation Bisimulation. In addition to this we give various lemmas which we will use later, and also the substitutivity property for the auxiliary relation. This last lemma will be used together with the equivalence of Evaluation Bisimulation and the auxiliary relation to show that Evaluation Bisimiation is a congruence relation, and not just an equivalence relation as already proved in chapter 10. In chapter 12 we then prove that the auxiliary relation is closed under evaluation and that the transitive closure is a symmetric relation. This will then enable us to deduce that the auxiliary relation is contained in Evaluation Bisimulation and hence that they are equal.

11.1 Auxiliary relation

Definition 11.1

For any, possibly open, CML$_\nu$ processes, e and f we define a relation $e \simeq^* f$ inductively using the rules given in figures 11.1 and 11.2. We call this relation the auxiliary relation to Evaluation Bisimulation, but we normally refer to this as the auxiliary relation.

$$\text{(NULL*)}\ \frac{}{\Lambda \simeq^* f}\ \text{if}\ \Lambda \simeq^o f \qquad \text{(VAR*)}\ \frac{}{x \simeq^* f}\ \text{if}\ x \simeq^o f,\ x \in \text{Var}$$

$$\text{(IN*)}\ \frac{}{n? \simeq^* f}\ \text{if}\ n? \simeq^o f \qquad \text{(LIT*)}\ \frac{}{l \simeq^* f}\ \text{if}\ l \simeq^o f,\ l \in \text{Lit}$$

$$\text{(UNIT*)}\ \frac{}{() \simeq^* f}\ \text{if}\ () \simeq^o f \qquad \text{(CON*)}\ \frac{e \simeq^* e'}{\mathbf{c}e \simeq^* f}\ \text{if}\ \mathbf{c}e' \simeq^o f$$

$$\text{(DEL*)}\ \frac{e \simeq^* e'}{[e] \simeq^* f}\ \text{if}\ [e'] \simeq^o f \qquad \text{(RES*)}\ \frac{e \simeq^* e'}{\nu n.e \simeq^* f}\ \text{if}\ \nu n.e' \simeq^o f$$

$$\text{(STRUC*)}\ \frac{e' \simeq^* f}{e \simeq^* f}\ \text{if}\ e \equiv e' \qquad \text{(OUT*)}\ \frac{e \simeq^* e'}{n!e \simeq^* f}\ \text{if}\ n!e' \simeq^o f$$

$$\text{(PAIR*)}\ \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2'}{(e_1, e_2) \simeq^* f}\ \text{if}\ (e_1', e_2') \simeq^o f$$

$$\text{(vPAIR*)}\ \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2'}{< e_1, e_2 > \simeq^* f}\ \text{if}\ < e_1', e_2' > \simeq^o f$$

$$\text{(LET*)}\ \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2'}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \simeq^* f}\ \text{if}\ \textbf{let}\ x = e_1'\ \textbf{in}\ e_2' \simeq^o f$$

$$\text{(APP*)}\ \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2'}{e_1 e_2 \simeq^* f}\ \text{if}\ e_1' e_2' \simeq^o f$$

Figure 11.1: Auxiliary Relation for Evaluation Bisimulation — Part 1

$$\text{(PAR*)}\quad \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2'}{e_1|e_2 \simeq^* f} \qquad \text{if } e_1'|e_2' \simeq^o f$$

$$\text{(WRAP*)}\quad \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2'}{e_1 \Rightarrow e_2 \simeq^* f} \qquad \text{if } e_1' \Rightarrow e_2' \simeq^o f$$

$$\text{(SUM*)}\quad \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2'}{e_1 \oplus e_2 \simeq^* f} \qquad \text{if } e_1' \oplus e_2' \simeq^o f$$

$$\text{(IF*)}\quad \frac{e_1 \simeq^* e_1' \quad e_2 \simeq^* e_2' \quad e_3 \simeq^* e_3'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \simeq^* f} \qquad \text{if } \textbf{if } e_1' \textbf{ then } e_2' \textbf{ else } e_3' \simeq^o f$$

$$\text{(FIX*)}\quad \frac{e \simeq^* e'}{\textbf{fix}\,(x = \textbf{fn} y \Rightarrow e) \simeq^* f} \qquad \text{if } \textbf{fix}\,(x = \textbf{fn} y \Rightarrow e') \simeq^o f$$

Figure 11.2: Auxiliary Relation for Evaluation Bisimulation — Part 2

Definition 11.2

As in definition 4.2 we define a derivation of $e \simeq^* f$ to be *normal* if the (STRUC) rule is used at most as the last rule and it is *simple* if it does not use the (STRUC) rule at all. We say that $e \simeq^* f$ is normal/simple if the derivation we use is normal/simple.

Lemma 11.3

For CML_ν expressions e and f then if $e \simeq^* f$, there exists a normal derivation for it.

Proof

The (STRUC) rule may be "pushed down" through all the other rules since $\equiv$ is a congruence. Two sequential usages of the (STRUC) rule may be unified since $\equiv$ is transitive. □

11.2 Initial properties of the auxiliary relation

Having now defined the auxiliary relation for processes and values, we prove various useful lemmas that we will use many times in the next chapter. We

start by showing that Evaluation Bisimulation is contained in the auxiliary relation.

Lemma 11.4

For CML$_\nu$ expressions e, f and g then

1. $e \simeq^* f \ \& \ f \simeq^o g \Rightarrow e \simeq^* g$
2. $e \simeq^* e$
3. $e \simeq^o f \Rightarrow e \simeq^* f$

Proof

1. We use the transitivity of $\simeq$ and the fact that we may assume that $e \simeq^* f$ is normal.
2. $e \simeq^o e$ and then use induction on the structure of e.
3. Follows from parts 1 & 2.

□

We now show the main result of this chapter, which is that the Auxiliary relation has a substitutivity property. This will be used to show that the Evaluation relation is a congruence and not just an equivalence relation.

Lemma 11.5 *Substitutivity property for* $\simeq^*$

For CML$_\nu$ expressions e, f, $e'(x)$ and $f'(x)$ then $e \simeq^* f$ and $e'(x) \simeq^* f'(x)$ implies that $e'[e/x] \simeq^* f'[f/x]$.

Proof

This follows by induction on the derivation of $e'(x) \simeq^* f'(x)$ using the fact that if $e'(x) \simeq^o f'(x)$ then $e'[f/x] \simeq^o f'[f/x]$. □

Finally we show that the transitive closure of the Auxiliary relation is symmetric. We do this for the same reason and use the same method as in section 4.4 on page 60.

Lemma 11.6

For CML$_\nu$ expressions e and f, $e \simeq^* f$ implies that $f \simeq^{*tc} e$.

Proof

This follows by tedious case splitting in the same way as lemma 4.11. □

Chapter 12

Congruence of Evaluation Bisimulation

We proceed in a similar manner to the Howe-style proof in chapter 4. We use the Auxiliary relation given in the previous chapter.

Having already shown in chapter 11 that $\simeq^o$ is a subset of $\simeq^*$, and that $\simeq^{*tc}$ is symmetric, we now show that $\simeq^{*tc}$ is closed under evaluation. We proceed using the transition relation. Once we have shown that $\simeq^{*tc}$ is closed under transition, we use theorem 9.29 to show that the Auxiliary relation is closed under evaluation. This method has various advantages. One is that it means we may split up the proof into various smaller sections, without having cyclic dependencies between lemmas. Another is that we do not have to deal with Evaluation Contexts. These contexts can be arbitrarily large, which will introduce many extra steps. In particular we do not have a result yet that says that for two CML_ν expressions e and f and a name n then $e \simeq f$ implies that $\nu n.e \simeq \nu n.f$. Without such a result the induction becomes even more complicated and long-winded.

We will prove all the results in this chapter with the restriction that all transitions/evaluations are on *closed* expressions. We may use this restriction since Evaluation Bisimulation is defined for closed expressions and we already have a substitutivity result, lemma 11.5, for the Auxiliary relation.

We start by considering those transitions which produce any of an input, an output or a value being returned. We first prove the result for simple

transitions and then extend the result to the general case using lemma 9.10. We will then do the same for transitions which involve a τ action.

Lemma 12.1

For CML$_\nu$ expressions e and f and a concretion C then $e \searrow \surd C$ and $e \simeq^* f$ simple imply that there is a concretion D with $f \Downarrow \surd D$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

Proof

We will proceed by induction on the derivation of $e \searrow \nu\vec{n}.\surd v.e'$. The last rule used:

VAL Hence $e = v, \vec{n} = \emptyset$ and $e' = \Lambda$. We now case split again on the form of v:

Literal Hence $v = l$ and $l \simeq^* f$. Therefore $l \simeq^o f$. Hence $f \Downarrow \surd v'.f'$ and for any abstraction A, $A\bullet < v, \Lambda > \simeq^o A\bullet < v', f' >$. Therefore by lemma 11.4 $A\bullet < v, \Lambda > \simeq^* A\bullet < v', f' >$ for any abstraction A.

Recursive function Hence $v = \mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}e_1)$ and there is an f_1 such that $e_1 \simeq^* f_1$ and $\mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}f_1) \simeq^o f$. Hence $f \Downarrow \surd v'.f'$ and for any abstraction A, $A\bullet < v, \Lambda > \simeq^o A\bullet < v', f' >$. Therefore by lemma 11.4(1) we get $A\bullet < v, \Lambda > \simeq^* A\bullet < v', f' >$ for any abstraction A.

Evaluated Pair Hence $v = < v_1, v_2 >$ and there are v_1', v_2' with $v_i \simeq^* v_i'$ and $< v_1', v_2' > \simeq^o f$. ($v_1', v_2'$ are values because otherwise $< v_1', v_2' >$ would not be correct syntactically.) But $< v_1', v_2' > \Downarrow \surd < v_1', v_2' > .\Lambda$ and hence there are v' and f' with $f \Downarrow v'.f'$ and for any A, $A\bullet << v_1', v_2' >, \Lambda > \simeq^o A\bullet < v', f' >$. But, by lemma 11.5, $A\bullet < v, \Lambda > \simeq^* A\bullet << v_1', v_2' >, \Lambda >$, and so by lemma 11.4 $A\bullet < v, \Lambda > \simeq^* A\bullet < v', f' >$, for any A.

Guarded expression Hence $v = [e_1]$ and there is an f_1 with $e_1 \simeq^* f_1$ and $[f_1] \simeq^o f$. Hence $f \Downarrow \surd v'.f'$ and for any abstraction A, $A\bullet < [f_1], \Lambda > \simeq^* A\bullet < v', f' >$. But, by lemma 11.5 and (DEL*), $A\bullet < v, \Lambda > \simeq^* A\bullet < [f_1], \Lambda >$, and so by lemma 11.4 $A\bullet < v, \Lambda > \simeq^* A\bullet < v', f' >$, for any A.

Therefore, with $D = v'.f'$, we have $f \Downarrow \surd D$ with $A \bullet C \simeq^* A \bullet D$, as required.

WRAP This case cannot occur since e_1 cannot be a guarded expression.

SUM$_1$ This case cannot occur since e_1 cannot be a guarded expression.

SUM$_2$ This case cannot occur since e_1 cannot be a guarded expression.

STRUC Hence $e \equiv e_1$ and $e' \equiv e_1'$. Hence $e_1 \searrow \nu\vec{n}.\surd v.e_1'$. Then using (STRUC*) we get $e_1 \simeq^* f$ and hence by induction hypothesis $f \Downarrow \surd D$ and $A \bullet \nu\vec{n} < v, e_1' > \simeq^{*tc} A \bullet D$ for any abstraction A. But $e' \equiv e_1'$ implies that $A \bullet C = A \bullet \nu\vec{n} < v, e' > \equiv A \bullet < v, e_1' >$ and hence using (STRUC*) $A \bullet C \simeq^{*tc} A \bullet D$.

uRES$_\surd$ Hence $e = \nu m.e''$ with $e'' \searrow \nu\vec{n}.\surd v.e'''$, where $e' = \nu m.e'''$. The last rule used in the derivation of $e \simeq^* f$ must have been (RES*) so there is an f'' with $e'' \simeq^* f''$ and $\nu m.f'' \simeq^o f$. Therefore by induction hypothesis there are $\vec{m}'$, v' and f' with $f'' \Downarrow \nu\vec{m}'.v''.f'''$ and for any abstraction A we know that $A \bullet \nu\vec{n} < v, e''' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', f''' >$. We restrict our attention to those abstractions A that do not have m as a free name.[1] Therefore, by using lemma 11.5, we see that $\nu m.(A \bullet \nu\vec{n} < v, e'' >) \simeq^{*tc} \nu m.(A \bullet \nu\vec{m}' < v'', f''' >)$. We deduce that $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} \nu m.(A \bullet \nu\vec{m}' < v'', f''' >)$ by noting that m is not free in A or v. We now case split on whether m is in the free names of v''.

[$m \in \mathrm{fn}(v)''$] Thus $\nu m.f'' \Downarrow \nu m\vec{m}'.\surd v''.f'''$. So there are $\vec{m}''$, v' and f' with $f \Downarrow \nu\vec{m}''.\surd v'.f'$ and $A \bullet \nu m\vec{m}' < v'', f''' > \simeq^o A \bullet \nu\vec{m}'' < v'.f' >$ for any abstraction, e.g. A. Because $m \in \mathrm{fn}(v)''$ we may deduce that $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu m\vec{m}' < v'', f''' >$. Therefore using lemma 11.4, $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}'' < v'.f' >$ as required.

[$m \notin \mathrm{fn}(v)''$] Thus $\nu m.f'' \Downarrow \nu\vec{m}'.\surd v''.\nu m.f'''$. Therefore there are $\vec{m}''$, v' and f' with $f \Downarrow \nu\vec{m}''.\surd v'.f'$ and for any abstraction, e.g. A, $A \bullet \nu\vec{m}' < v'', \nu m.f''' > \simeq^o A \bullet \nu\vec{m}'' < v'.f' >$. But we can deduce

[1] We don't strictly need to do this. To get around this restriction we can α-convert e and f so that m is not a bound name, then work through the proof and then at the end α-convert it back.

that $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', \nu m.f''' >$ since $m \notin \mathrm{fn}(v)''$. Therefore $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}'' < v'.f' >$ as required, using lemma 11.4.

rRES$_\surd$ Hence $e = \nu m.e''$ with $e'' \searrow \nu\vec{n}'.\surd v.e'$, for some process e'' where $\vec{n}' = \vec{n} \setminus \{m\}$. The last rule using in the derviation of $e \simeq^* f$ must have been (RES*) so there is an f'' with $e'' \simeq^* f''$ and $\nu m.f'' \simeq^o f$. Therefore by induction hypothesis there are $\vec{m}'$, v' and f' with $f'' \Downarrow \nu\vec{m}'.v''.f'''$ and for any A we know that $A \bullet \nu\vec{n}' < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', f''' >$. We will restrict our attention to those A's that do not have m as a free name. Therefore by using lemma 11.5 on page 180 we may deduce that $\nu m.(A \bullet \nu\vec{n}' < v, e' >) \simeq^{*tc} \nu m.(A \bullet \nu\vec{m}' < v'', f''' >)$. We now case split on whether m is in the free names of v''.

[$m \in \mathrm{fn}(v'')$] Therefore $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu m\vec{m}' < v'', f''' >$, because m is not a free name in A. Therefore we also know that $\nu m.f'' \Downarrow \nu m\vec{m}'.v''.f'''$ and therefore there are $\vec{m}''$, v' and f' with $f \Downarrow \nu\vec{m}''.v'.f'$ and for any abstraction, e.g. A, we also know that $A \bullet \nu m\vec{m}' < v'', f''' > \simeq^o \nu\vec{m}'' < v', f' >$. Therefore, by lemma 11.4, $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}'' < v', f' >$, as required.

[$m \notin \mathrm{fn}(v'')$] Therefore $\nu m.f'' \Downarrow \nu\vec{m}'.v''.\nu m.f'''$. Hence by induction hypothesis there are $\vec{m}''$, v' and f' with $\nu m.f'' \Downarrow \nu\vec{m}''.v'.f'$ and for any abstraction, e.g. A, $A \bullet \nu\vec{m}' < v'', \nu m.f''' > \simeq^o \nu\vec{m}'' < v', f' >$. But $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', \nu m.f''' >$, since m is not a free name in A or v''. Therefore we use lemma 11.4 to derive $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} \nu\vec{m}'' < v', f' >$, as required.

PAR$_1$ Hence $e = e_1|e_2$ and there is an e_2' with $e_2 \searrow \nu\vec{n}.\surd v.e_2'$ and $e' = e_1|e_2'$. Therefore the last rule used in the derivation of $e \simeq^* f$ must have been (PAR*) and so there must be f_1 and f_2 with $e_1 \simeq^* f_1$, $e_2 \simeq^* f_2$ and $f_1|f_2 \simeq^o f$. By induction hypothesis there are $\vec{m}$, v'' and f_2' with $f_2 \Downarrow \nu\vec{m}.\surd v''.f_2'$ and for any abstraction A we may observe that $A \bullet \nu\vec{n} < v, e_2' > \simeq^{*tc} A \bullet \nu\vec{m} < v'', f_2' >$. Using PAR$_1$ we see that $f_1|f_2 \Downarrow \nu\vec{m}.\surd v''.(f_1|f_2')$[2] and so there are $\vec{m}'$, v' and f' with

[2] We assume w.l.o.g. that $\vec{m} \cap \mathrm{fn}(f_1) = \emptyset$. If this is not the case then we α-convert to ensure

$f \Downarrow \nu\vec{m}'.\surd v'.f'$ and that for any abstraction, e.g. A, we know that $A \bullet \nu\vec{m} < v'', f_1|f_2' > \simeq^o A \bullet \nu\vec{m}' < v'.f' >$. But $\vec{m} \cap \mathrm{fn}(f_1) = \emptyset$ and lemma 10.21 implies that $f_1|(A \bullet \nu\vec{m} < v'', f_2' >) \simeq^o A \bullet \nu\vec{m}' < v'.f' >$.

By lemma 11.5 $e_1|(A \bullet \nu\vec{n} < v, e_2' >) \simeq^{*tc} f_1|(A \bullet \nu\vec{m} < v'', f_2' >)$ and then we derive $A \bullet \nu\vec{n} < v, e_1|e_2' > \simeq^{*tc} f_1|(A \bullet \nu\vec{m} < v'', f_2' >)$ using lemmas 10.21 and 11.4.[3] This plus the last equation in the previous paragraph gives $A \bullet \nu\vec{n} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'.f' >$, as required.

No other cases can occur. □

Lemma 12.2

For CML$_\nu$ expressions e and f, name n and a concretion C then $e \searrow n!C$ and $e \simeq^* f$ simple imply that there is a concretion D with $f \Downarrow n!D$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

Proof

If $C = \nu\vec{m} < v, e' >$ then we will proceed by induction on the derivation of $e \searrow \nu\vec{m}.n!v.e'$. Last rule used:

OUT This follows in exactly the same way as the VAL case above.

WRAP So $e = e_1 \Rightarrow e_2$ and there is an e_1' with $e_1 \searrow \nu\vec{m}.n!v.e_1'$ and $e' = e_2 e_1'$. The last rule used in the derivation of $e \simeq^* f$ must have been (WRAP*) and so there are f_1 and f_2 with $e_1 \simeq^* f_1$, $e_2 \simeq^* f_2$ and $f_1 \Rightarrow f_2 \simeq^o f$. By induction hypothesis there are $\vec{m}'$, v'' and f_1' with $f_1 \Downarrow \nu\vec{m}'.n!v''.f_1'$ and for any abstraction A we have $A \bullet \nu\vec{m} < v, e_1' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', f_1' >$. Using WRAP implies that $f_1 \Rightarrow f_2 \Downarrow \nu\vec{m}'.n!v''.f_2 f_1'$.[4] This in turn implies that there are $\vec{m}''$, v' and f' with $f \Downarrow \nu\vec{m}''.n!v'.f'$ and for any abstraction, e.g. A, $A \bullet \nu\vec{m}' < v'', f_2 f_1' > \simeq^o A \bullet \nu\vec{m}'' < v', f' >$. But by lemmas 10.28 and 11.4 we observe that $A \bullet \nu\vec{m} < v, e_2 e_1' > \simeq^* e_2(A \bullet \nu\vec{m} < v, e_1' >)$. Lemma 11.5 implies that $e_2(A \bullet \nu\vec{m} < v, e_1' >) \simeq^{*tc} f_2(A \bullet \nu\vec{m}' < v'', f_1' >)$. Lemma 10.28 implies that $f_2(A \bullet \nu\vec{m}' < v'', f_1' >) \simeq^o A \bullet \nu\vec{m}'' < v', f' >$. Putting the last three equations together, together with lemma 11.4, gives $A \bullet \nu\vec{m} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}'' < v', f' >$, as required.

---

that this is the case.

[3] Notice that $\vec{n} \cap \mathrm{fn}(e_1) = \emptyset$.

[4] We again assume w.l.o.g. that $\vec{m}' \cap \mathrm{fn}(f_2)$. If this is not the case then we α-convert f_1 and f_1' to ensure that this requirement holds.

SUM$_1$ Hence $e = e_1 \oplus e_2$ and $e_1 \searrow \nu\vec{m}.n!v.e'$. The last rule used in the derivation of $e \simeq^* f$ must have been (SUM*) and so there are f_1 and f_2 with $e_1 \simeq^* f_1$, $e_2 \simeq^* f_2$ and $f_1 \oplus f_2 \simeq f$. By induction hypothesis there are $\vec{m}'$, v'' and f_1' with $f_1 \Downarrow \nu\vec{m}'.n!v''.f_1'$ and for any abstraction A we have $A \bullet \nu\vec{m} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', f_1' >$. By lemma 9.28 and theorem 9.29 twice there are $\vec{m}'''$, v''' and f_1''' with $\vec{m}''' \cap \mathrm{fn}(f_2) = \emptyset$, $\vec{m}'''$ and $\vec{m}'$ being in a 1-1 mapping, $v''' = v''[\vec{m}'''/\vec{m}']$, $f_1''' = f_1'''[\vec{m}'''/\vec{m}']$ and $f_1 \Downarrow \nu\vec{m}'''.n!v'''.f_1'''$. But $A \bullet \nu\vec{m}' < v'', f_1' > \equiv_\alpha A \bullet \nu\vec{m}''' < v''', f_1''' >$ implies that $A \bullet \nu\vec{m} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}''' < v''', f_1''' >$ using (STRUC*). Using SUM$_1$ we may derive that $f_1 \oplus f_2 \Downarrow \nu\vec{m}'''.n!v'''.f_1'''$. This implies that there are $\vec{m}''$, v' and f' with $f \Downarrow \nu\vec{m}''.n!v'.f$ and for any abstraction, in particular A, $A \bullet \nu\vec{m}''' < v''', f_1''' > \simeq^{*tc} A \bullet \nu\vec{m}'' <'', f' >$. Therefore by lemma 11.4 we get $A \bullet \nu\vec{m} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}'' < v', f' >$, as required.

SUM$_2$ Follows in the same way as SUM$_1$ above.

STRUC This follows in exactly the same way as the STRUC case above.

rRES$_!$ This follows in exactly the same way as the rRES$_\surd$ case above.

uRES$_!$ This follows in exactly the same way as the uRES$_\surd$ case above.

PAR$_1$ This follows in exactly the same way as the PAR$_1$ case above.

PAR$_2$ Hence $e = e_1|e_2$ and there is an e_1' with $e_1 \searrow \nu\vec{m}.n!v.e_1'$ and $e' = e_1'|e_2$. Therefore the last rule used in the derivation of $e \simeq^* f$ must have been (PAR*) and so there are f_1 and f_2 with $e_1 \simeq^* f_1$, $e_2 \simeq^* f_2$ and $f_1|f_2 \simeq^o f$. By induction hypothesis there are $\vec{m}'$, v'' and f_1' with $f_1 \Downarrow \nu\vec{m}.n!v''.f_1'$ and $A \bullet \nu\vec{m} < v, e_1' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', f_1' >$ for any abstraction A. Using PAR$_2$ we see that $f_1|f_2 \Downarrow \nu\vec{m}'.n!v''.(f_1'|f_2)$[5] and so there are $\vec{m}'$, v' and f' with $f \Downarrow \nu\vec{m}''.n!v'.f'$ and, for any abstraction, e.g. A, we know that $A \bullet \nu\vec{m}' < v'', f_1'|f_2 > \simeq^o A \bullet \nu\vec{m}'' < v'.f' >$. But $\vec{m}' \cap \mathrm{fn}(f_2) = \emptyset$ implies that $(A \bullet \nu\vec{m}' < v'', f_1' >)|f_2 \simeq^o A \bullet \nu\vec{m}'' < v'.f' >$. Lemma 11.5 implies that $(A \bullet \nu\vec{m} < v, e_1' >)|e_2 \simeq^{*tc} (A \bullet \nu\vec{m}' < v'', f_1' >)|f_2$. Then (STRUC*)

[5] We assume w.l.o.g. that $\vec{m} \cap \mathrm{fn}(f_2) = \emptyset$. If this is not the case then we α-convert to ensure that this is the case.

and lemma 11.4 imply that $A \bullet \nu\vec{m} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}'' < v'.f' >$, as required.

CON$_\alpha$ Hence $e = \mathbf{c}e_1$, $e' = \mathbf{c}e_1'$ and $e_1 \searrow \nu\vec{m}.n!v.e_1'$. The last rule in the derivation of $e \simeq^* f$ must have been (CON*) so there is an f_1 with $e_1 \simeq^* f_1$ and $\mathbf{c}f_1 \simeq^o f$. Therefore by induction hypothesis there are $\vec{m}'$, v'' and f_1' with $f_1 \Downarrow \nu\vec{m}'.n!v''.f_1'$ and $A \bullet \nu\vec{m} < v, e_1' > \simeq^{*tc} A \bullet \nu\vec{m}' < v'', f_1' >$ for any abstraction A. Then lemmas 11.5, 10.28 and 11.4 imply that $A \bullet \nu\vec{m} < v, \mathbf{c}e_1' > \simeq^{*tc} \mathbf{c}(A \bullet \nu\vec{m}' < v'', f_1' >)$.

CON$_\alpha$ and $f_1 \Downarrow \nu\vec{m}'.n!v''.f_1'$ imply that $\mathbf{c}f_1 \Downarrow \nu\vec{m}'.n!v''.\mathbf{c}f_1'$. Therefore there are $\vec{m}''$, v' and f' with $f \Downarrow \nu\vec{m}''.n!v'.f'$ and for any abstraction, e.g. A, $A \bullet \nu\vec{m}' < v'', \mathbf{c}f_1' > \simeq^o A \bullet \nu\vec{m}'' < v', f' >$. But lemma 10.28 implies that $\mathbf{c}(A \bullet \nu\vec{m}' < v'', f_1' >) \simeq^o A \bullet \nu\vec{m}'' < v', f' >$ and lemma 11.4 gives $A \bullet \nu\vec{m} < v, e' > \simeq^{*tc} A \bullet \nu\vec{m}'' < v', f' >$, as required.

The APP$_\alpha$, PAIR$_\alpha$, IF$_\alpha$ and LET$_\alpha$ cases all follow in the same was as the CON$_\alpha$ case. No other cases can occur. □

Lemma 12.3

For CML$_\nu$ expressions e and f, name n and an abstraction A then $e \searrow n?A$ and $e \simeq^* f$ simple imply that there is an abstraction B with $f \Downarrow n?B$ and for any concretion C we may deduce that $C \bullet A \simeq^{*tc} C \bullet B$.

Proof

If $A = (x, e')$ then we will proceed by induction on the derivation of $e \searrow n?x.e'$. Last rule used:

IN Therefore $e = n?$ and $e' = \Lambda$. So the last rule used in the derivation of $e \simeq^* f$ must have been (IN*). Therefore $e \simeq^o f$ and so there is an f' with $f \Downarrow n?x.f'$ and $C \bullet (x, \Lambda) \simeq^* C \bullet (x, f')$ for any concretion C, as required.

RES$_?$ Follows in the same way as the uRES$_!$ case above.

All other cases follow in the same way as the corresponding case in lemma 12.2 or else do not occur. □

Lemma 12.4

For CML$_\nu$ expressions e and f with $e \simeq^* f$ then:

1. For concretion C with $e \searrow \sqrt{C}$ then there is a concretion D with $f \Downarrow \sqrt{D}$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

2. For concretion C and name n with $e \searrow n!C$ then there is a concretion D with $f \Downarrow n!D$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

3. For abstraction A and name n with $e \searrow n?A$ then there is an abstraction B with $f \Downarrow n?B$ and for any concretion C we may deduce that $C \bullet A \simeq^{*tc} C \bullet B$.

Proof

All three parts follow in the same way. By lemma 11.3 we may assume that the derivation of $e \simeq^* f$ is normal. If it is simple the result follows by lemmas 12.1, 12.2 and 12.3. Otherwise, and we just give the first part as an example, the last rule used was (STRUC*) and so there is an e' with $e \equiv e'$ and $e' \searrow \sqrt{C}$. Therefore, using lemma 12.1, there is a concretion D with $f \Downarrow \sqrt{D}$ and for any abstraction A we get $A \bullet C \simeq^{*tc} A \bullet D$ as required. The other two cases follow similarly. □

Lemma 12.5

For CML$_\nu$ expressions e_1, e_2, f_1 and f_2 with $e_1 \simeq^* f_1$ and $e_2 \simeq^* f_2$ then $e_1 \searrow \nu\vec{m}.\sqrt{v}.e_1'$, for some $\vec{m}$, v and e_1', implies that $f_1 \searrow \nu\vec{m}'.\sqrt{v'}.f_1'$, for some $\vec{m}'$, v' and f_1', with:

- $\nu\vec{m}.(e_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$.
- $v = \mathbf{true}$ implies that $v' = \mathbf{true}$ and $e_1'|e_2 \simeq^{*tc} f_1'|f_2$.
- $v = \mathbf{false}$ implies that $v' = \mathbf{false}$ and $e_1'|e_2 \simeq^{*tc} f_1'|f_2$.
- $\nu\vec{m}.(e_1'|\mathbf{let}\ x = e_2\ \mathbf{in} < v, x >) \simeq^{*tc} \nu\vec{m}'.(f_1'|\mathbf{let}\ x = f_2\ \mathbf{in} < v', x >)$.
- $v = \mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow e_3)$ implies that $v' = \mathbf{fix}\,(x = \mathbf{fn} y \Rightarrow f_3)$ and $\nu\vec{m}.(e_1'|\mathbf{let}\ y = e_2\ \mathbf{in}\ e_3[v/x]) \simeq^{*tc} \nu\vec{m}'.(f_1'|\mathbf{let}\ y = f_2\ \mathbf{in}\ f_3[v'/x])$.
- $\nu\vec{m}.(e_1'|e_2[v/x]) \simeq^{*tc} \nu\vec{m}'.(f_1'|f_2[v'/x])$.

Proof

Each case follows in essentially the same way. We give the first as an example since it is the hardest.

For the first case we start by assuming that the derivation of $e_1 \simeq^* f_1$ is simple. We proceed by induction on the derivation of $e_1 \searrow \nu\vec{m}.\surd v.e_1'$. Last rule used:

VAL Hence $e_1 = v$, $\vec{m} = \emptyset$ and $e_1' = \Lambda$. The last rule used in the derivation of $e_1 \simeq^* f_1$ must have been (VAR*), (DEL*), (LIT*), (UNIT*), (vPAIR*) or (FIX*). If the last rule use was not (DEL*) then the result follows from lemma 10.32. Otherwise $e_1 = [g_1]$ and there is a g_2 with $g_1 \simeq^* g_2$ and $[g_2] \simeq^o f_1$. But $[g_2] \Downarrow \surd[g_2].\Lambda$ so there must be $\vec{m}'$, v' and f_1' with $f_1 \Downarrow \nu\vec{m}'.\surd v'.f_1'$ and for any abstraction A we know that $A\bullet < [g_2], \Lambda > \simeq^o A \bullet \nu\vec{m}' < v', f_1' >$. Therefore, using lemma 10.32, we may deduce that $\delta(\mathbf{c}, [g_2]) \simeq^o \nu\vec{m}'.(f'|\delta(\mathbf{c}, f'))$. We now case split on what c is. It can only be sync or always for $\delta(\mathbf{c}, [g_2])$ to be defined.

- **sync** Therefore $\delta(\mathbf{sync}, [g_2]) = g_2$. But $\nu\vec{m}.(e_1'|\delta(\mathbf{sync}, v)) \equiv g_1$ and $g_1 \simeq^* g_2$. Therefore, again using lemma 11.4, we may show that $\nu\vec{m}.(e_1'|\delta(\mathbf{sync}, v)) \simeq^* \nu\vec{m}'.(f'|\delta(\mathbf{c}, f'))$, as required.
- **always** Therefore $\delta(\mathbf{always}, [g_2]) = [g_2]$. But $\nu\vec{m}.(e_1'|\delta(\mathbf{always}, v)) \equiv [g_1]$ and $g_1 \simeq^* g_2$. Therefore by lemma 11.5 we see that $[g_1] \simeq^* [g_2]$. So lemma 11.4 gives $\nu\vec{m}.(e_1'|\delta(\mathbf{sync}, v)) \simeq^* \nu\vec{m}'.(f'|\delta(\mathbf{c}, f'))$, as required.

STRUC Hence $e_1 \equiv g_1$ and $e_1' \equiv g_1'$ with $g_1 \searrow \nu\vec{n}.\surd v.g_1'$. But $g_1 \simeq^* f_1$ and so, by induction hypothesis, there are $\vec{m}'$, v' and f_1' with $f_1 \Downarrow \nu\vec{m}'.\surd v'.f_1'$ and $\nu\vec{m}.(g_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$. But $\nu\vec{m}.(e_1'|\delta(\mathbf{c}, v)) \equiv \nu\vec{m}.(g_1'|\delta(\mathbf{c}, v))$ and so by (STRUC*) we obtain $\nu\vec{m}.(e_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$, as required.

uRES$_\surd$ Hence $e_1 = \nu n.g_1$ and $g_1 \searrow \nu\vec{m}.\surd v.g_1'$, where $\nu n.g_1' = e_1'$. Last rule used in derivation of $e_1 \simeq^* f_1$ must have been (RES*) and so there is a g_2 with $g_1 \simeq^* g_2$ and $\nu n.g_2 \simeq^o f_1$. By induction hypothesis there are $\vec{m}''$, v'' and g_2' with $g_2 \Downarrow \nu\vec{m}''.\surd v''.g_2'$ and $\nu\vec{m}.(g_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}''.(g_2'|\delta(\mathbf{c}, v''))$. We now case split on whether $n \in \mathrm{fn}(()v'')$:

[$n \in \mathrm{fn}(v'')$] Therefore $\nu n.g_2 \Downarrow \nu n\vec{m}''.\surd v''.g_2'$ and $f_1 \Downarrow \nu\vec{m}'.\surd v'.f_1'$ for some $\vec{m}'$, v' and f_1' and $A \bullet \nu n\vec{m}'' < v'', g_2' > \simeq^o A \bullet \nu\vec{m}' < v', f_1' >$ for any abstraction A. Therefore, using lemma 10.32, we see that $\nu n\vec{m}''.(g_2'|\delta(\mathbf{c}, v'')) \simeq^o \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$. But, using lemma 11.5 and (STRUC*) we see that $\nu\vec{m}.(\nu n.g_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu n\vec{m}''.(g_2'|\delta(\mathbf{c}, v''))$. So by lemma 11.4 $\nu\vec{m}.(\nu n.g_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$, as required.

[$n \notin \mathrm{fn}(v'')$] Therefore $\nu n.g_2 \Downarrow \nu\vec{m}''.\surd v''.\nu n.g_2'$ and $f_1 \Downarrow \nu\vec{m}'.\surd v'.f_1'$ for some $\vec{m}'$, v' and f_1' and $A \bullet \nu\vec{m}'' < v'', \nu n.g_2' > \simeq^o A \bullet \nu\vec{m}' < v', f_1' >$ for any abstraction A. Therefore, using lemma 10.32, we see that $\nu\vec{m}''.(\nu n.g_2'|\delta(\mathbf{c}, v'')) \simeq^o \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$. Using lemma 11.5 and (STRUC*) we see that $\nu\vec{m}.(\nu n.g_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}''.(\nu n.g_2'|\delta(\mathbf{c}, v''))$. Therefore, by lemma 11.4 $\nu\vec{m}.(\nu n.g_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$, as required.

rRES$_\surd$ Follows in essentially the same way as the uRES$_\surd$ case above.

PAR$_1$ Hence $e_1 = e_{11}|e_{12}$ for some e_{11} and e_{12}, with $e_{12} \searrow \nu\vec{m}.\surd v.e_{12}'$ and $e_1' = e_{11}|e_{12}'$. Therefore the last rule used in the derivation of $e_1 \simeq^* f_1$ must have been (PAR*) and so there are f_{11} and f_{12} with $e_{11} \simeq^* f_{11}$, $e_{12} \simeq^* f_{12}$ and $f_{11}|f_{12} \simeq^o f_1$. By induction hypothesis there are $\vec{m}''$[6], v'' and f_{12}' with $f_{12} \Downarrow \nu\vec{m}''.\surd v''.f_{12}'$ and $\nu\vec{m}.(e_{12}'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}''.(f_{12}'|\delta(\mathbf{c}, v''))$. PAR$_1$ implies that $f_{11}|f_{12} \Downarrow \nu\vec{m}''.\surd v''.(f_{11}|f_{12}')$ and so there are $\vec{m}'$, v' and f_1' with $f_1 \Downarrow \nu\vec{m}'.\surd v'.f_1'$ and $A \bullet \nu\vec{m}'' < v'', f_{11}|f_{12}' > \simeq^o A \bullet \nu\vec{m}'. < v', f_1' >$ for any abstraction A. So $\nu\vec{m}''.(f_{11}|f_{12}'|\delta(\mathbf{c}, v'')) \simeq^o \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$ using lemma 10.32. But $\nu\vec{m}.(e_{12}'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}''.(f_{12}'|\delta(\mathbf{c}, v''))$, $e_{11} \simeq^* f_{11}$, lemma 11.5 and (STRUC*) allow us to derive that $\nu\vec{m}.(e_{11}|e_{12}'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}''.(f_{11}|f_{12}'|\delta(\mathbf{c}, v''))$. Then lemma 11.4 show us that $\nu\vec{m}.(e_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$, as required.

If the derivation of $e_1 \simeq^* f_1$ was not simple then by lemma 9.10 there must be a normal derivation. There there is a g with $g \equiv e_1$ and $g \simeq^* f_1$ simple. But $g \searrow \nu\vec{m}.\surd v.e_1'$. Therefore using the simple case above, there are $\vec{m}'$, v' and f_1' with $f_1 \Downarrow \nu\vec{m}'.\surd v'.f_1'$ and $\nu\vec{m}.(e_1'|\delta(\mathbf{c}, v)) \simeq^{*tc} \nu\vec{m}'.(f_1'|\delta(\mathbf{c}, v'))$. □

[6] Additionally we require that $\vec{m}'' \cap \mathrm{fn}(f_{11}) = \emptyset$.

Lemma 12.6

For CML_ν expressions e, e' and f then $e \searrow \tau.e'$ and $e \simeq^* f$ simple imply that there is an expression f' with $f \searrow \tau^*.f'$ and $e' \simeq^{*tc} f'$.

Proof

We will use lemma 10.23 many times without reference. We proceed by induction on the derivation of $e \searrow \tau.e'$. Last rule used:

STRUC Hence there are e_1 and e_1' with $e \equiv e_1$, $e_1' \equiv e'$ and $e_1 \searrow \tau.e_1'$. Then using (STRUC*) we get $e_1 \simeq^* f$ and hence by induction hypothesis there is an f' with $f \searrow \tau^*.f'$ and $e_1' \simeq^{*tc} f'$. But $e_1' \equiv e'$ and (STRUC*) imply that $e' \simeq^{*tc} f'$, as required.

RES$_\tau$ Hence there are n, e_1 and e_1' with $e = \nu n.e_1$, $e' = \nu n.e_1'$ and $e_1 \searrow \tau.e_1'$. Last rule using in derivation of $e \simeq^* f$ must have been (RES*) and so there is an f_1 with $e_1 \simeq^* f_1$ and $\nu n.f_1 \simeq^o f$. By induction hypothesis there is an f_1' with $f_1 \searrow \tau^*.f_1'$ and $e_1' \simeq^{*tc} f_1'$. Then RES$_\tau$ implies that $\nu n.f_1 \searrow \tau.\nu n.f_1'$ and therefore there is an f' with $f \searrow \tau^*.f'$ and $\nu n.f_1' \simeq^o f'$. But lemma 11.5 and $e_1' \simeq^{*tc} f_1'$ imply that $\nu n.e_1' \simeq^{*tc} \nu n.f_1'$. Therefore lemma 11.4 implies that $e' \simeq^{*tc} f'$, as required.

PAR$_1$ Hence there are e_1, e_2 and e_2' with $e = e_1|e_2$, $e' = e_1|e_2'$ and $e_2 \searrow \tau.e_2'$. The last rule used in the derviation of $e \simeq^* f$ must have been (PAR*) and so there are f_1 and f_2 with $e_1 \simeq^* f_1$ and $e_2 \simeq^* f_2$. By induction hypothesis there is an f_2' with $f_2 \searrow \tau^*.f_2'$ and $e_2' \simeq^{*tc} f_2'$. PAR$_1$ implies that $f_1|f_2 \searrow \tau^*.f_1|f_2'$ and so there is an f' with $f \searrow \tau^*.f'$ and $f_1|f_2' \simeq^o f'$. But lemma 11.5 and $e_2' \simeq^{*tc} f_2'$ imply that $e_1|e_2' \simeq^{*tc} f_1|f_2'$. Therefore lemma 11.4 implies that $e' \simeq^{*tc} f'$, as required.

PAR$_2$ Hence there are e_1, e_2 and e_1' with $e = e_1|e_2$, $e' = e_1'|e_2$ and $e_1 \searrow \tau.e_1'$. The last rule used in the derviation of $e \simeq^* f$ must have been (PAR*) and so there are f_1 and f_2 with $e_1 \simeq^* f_1$ and $e_2 \simeq^* f_2$. By induction hypothesis there is an f_1' with $f_1 \searrow \tau^*.f_1'$ and $e_1' \simeq^{*tc} f_1'$. PAR$_2$ implies that $f_1|f_2 \searrow \tau^*.f_1'|f_2$ and so there is an f' with $f \searrow \tau^*.f'$ and $f_1'|f_2 \simeq^o f'$. But lemma 11.5 and $e_1' \simeq^{*tc} f_1'$ imply that $e_1'|e_2 \simeq^{*tc} f_1'|f_2$. Therefore lemma 11.4 implies that $e' \simeq^{*tc} f'$, as required.

CON$_\alpha$ Hence there are e_1 and e_1' with $e = \mathbf{c}e_1$, $e' = \mathbf{c}e_1'$ and $e_1 \searrow \tau.e_1'$. The last rule used in the derivation of $e \simeq^* f$ must have been (CON*) and so there is an f_1 with $e_1 \simeq^* f_1$ and $\mathbf{c}f_1 \simeq^o f$. Hence by induction hypothesis there is an f_1' with $f_1 \searrow \tau^*.f_1'$ and $e_1' \simeq^{*tc} f_1'$. CON$_\alpha$ implies that $\mathbf{c}f_1 \searrow \tau^*.\mathbf{c}f_1'$ and so there is an f' with $f \searrow \tau^*.f'$ and $\mathbf{c}f_1' \simeq^o f'$. But lemma 11.5 and $e_1' \simeq^{*tc} f_1'$ imply that $\mathbf{c}e_1' \simeq^{*tc} \mathbf{c}f_1'$. Therefore lemma 11.4 implies that $e' \simeq^{*tc} f'$, as required.

APP$_\surd$ Hence there are e_1, e_2 e_1', $v = \mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}e_3)$ and $\vec{n}$ with $e = e_1e_2$, $e' = \nu\vec{n}.(e_1'|\mathbf{let}\ y{=}e_2\ \mathbf{in}\ e_3[v/x])$ and $e_1 \searrow \nu\vec{n}.\surd v.e_1'$. Therefore the last rule used in the derivation of $e \simeq^* f$ must have been (APP*) and so there are f_1 and f_2 with $e_1 \simeq^* f_1$, $e_2 \simeq^* f_2$ and $f_1f_2 \simeq^o f$. Lemma 12.5 implies that there are $\vec{m}'$, v' and f_1' with $f_1 \searrow \tau^*.\nu\vec{m}'.\surd v'.f_1'$ and $v' = \mathbf{fix}\,(x{=}\mathbf{fn}y{\Rightarrow}f_3)$. We also see that $\nu\vec{m}.(e_1'|\mathbf{let}\ x{=}e_2\ \mathbf{in}\ e_3[v/x]) \simeq^{*tc} \nu\vec{m}'.(f_1'|\mathbf{let}\ x{=}f_2\ \mathbf{in}\ f_3[v'/x])$. But $f_1f_2 \searrow \tau^*.\nu\vec{m}'.(f_1'|\mathbf{let}\ x = f_2\ \mathbf{in}\ f_3[v'/x])$. Hence by lemma 10.23 we see that there is an f' with $f \searrow \tau^*.f'$ and $\nu\vec{m}'.(f_1'|\mathbf{let}\ x{=}f_2\ \mathbf{in}\ f_3[v'/x]) \simeq^o f'$. Therefore by lemma 11.4 we get $e' \simeq^{*tc} f'$, as required.

COM$_1$ Hence there are e_1, e_1', e_2, e_2', n, $\vec{m}$ and v with $e_1 \searrow \nu\vec{m}.n!v.e_1'$, $e_2 \searrow n?x.e_2'$ and $e' = \nu\vec{m}.(e_1'|e_2'[v/x])$. Therefore the last rule used in the derivation of $e \simeq^* f$ must have been (PAR*). So there are f_1 and f_2 with $e_1 \simeq^* f_1$, $e_2 \simeq^* f_2$ and $f_1|f_2 \simeq^o f$. Using lemma 12.4 twice we get f_1', f_2', $\vec{m}'$ and v' with $f_1 \Downarrow \nu\vec{m}'.n!v'.f_1'$, $f_2 \Downarrow n?x.f_2'$ and $\nu\vec{m}.(e_1'|e_2'[v/x]) \simeq^{*tc} \nu\vec{m}'.(f_1'|f_2'[v'/x])$. But $f_1|f_2 \searrow \tau^*.\nu\vec{m}'.(f_1'|f_2'[v'/x])$ and so, using lemma 10.23, there is an f' with $f \searrow \tau^*.f'$ and $\nu\vec{m}'.(f_1'|f_2'[v'/x]) \simeq^o f'$. Therefore by lemma 11.4 we get $e' \simeq^o f'$, as required.

COM$_2$ This follows in the symmetric way to COM$_1$ above.

The APP$_\alpha$, PAIR$_\alpha$, IF$_\alpha$ and LET$_\alpha$ cases all follow in the same was as the CON$_\alpha$ case. The CON$_\surd$, PAIR$_\surd$, IF$_t$, IF$_f$ and LET$_\surd$ cases all follow in the same was as the APP$_\surd$ case. □

Lemma 12.7

For CML$_\nu$ expressions e, e' and f then $e \searrow \tau.e'$ and $e \simeq^* f$ imply that there is an expression f' with $f \searrow \tau^*.f'$ and $e' \simeq^* f'$.

Proof

This follows in the same way as lemma 12.4. □

We may now prove the summary lemma, which states that the transitive closure of the auxiliary relationship is closed under evaluation. We may then, finally, prove that the auxiliary relation is the same as Evaluation Bisimulation. This will allow us, together with lemma 11.5, to show that Evaluation Bisimulation is indeed a congruence relation.

Lemma 12.8

For CML$_\nu$ expressions e and f with $e \simeq^* f$ then:

1. For concretion C with $e \Downarrow \surd C$ then there is a concretion D with $f \Downarrow \surd D$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

2. For concretion C and name n with $e \Downarrow n!C$ then there is a concretion D with $f \Downarrow n!D$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

3. For abstraction A and name n with $e \Downarrow n?A$ then there is an abstraction B with $f \Downarrow n?B$ and for any concretion C we may deduce that $C \bullet A \simeq^{*tc} C \bullet B$.

4. For concretion D with $f \Downarrow \surd D$ then there is a concretion C with $e \Downarrow \surd C$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

5. For concretion D and name n with $f \Downarrow n!D$ then there is a concretion C with $e \Downarrow n!C$ and for any abstraction A we may deduce that $A \bullet C \simeq^{*tc} A \bullet D$.

6. For abstraction B and name n with $f \Downarrow n?B$ then there is an abstraction A with $f \Downarrow n?A$ and for any concretion C we may deduce that $C \bullet A \simeq^{*tc} C \bullet B$.

Proof

The first three follow from theorem 9.29 and lemmas 12.4 and 12.7. The last three follow in the same way, additionally using lemma 11.6. □

Restatement of Theorem 10.5

Evaluation Bisimulation is a congruence relation.

Proof of Theorem 10.5

In chapter 10 we showed that $\simeq^o$ is an equivalence relation. Lemma 12.8 implies that $\simeq^{*tc} \subseteq \simeq^o$. But $\simeq^o \subseteq \simeq^*$ by lemma 11.4 and $\simeq^* \subseteq \simeq^{*tc}$ follows obviously. Hence we have $\simeq^o \subseteq \simeq^* \subseteq \simeq^{*tc} \subseteq \simeq^o$. Therefore $\simeq^o = \simeq^*$. The result then follows from lemma 11.5. □

Part III

Conclusion

Chapter 13

Conclusion

We have shown for two different calculi that it is possible to use an evaluation based method to describe the operational semantics of process calculi. We have also shown that weak applicative bisimulations may be defined and are shown to be congruence relations. However there are problems with using an evaluation relation. We now consider the advantages and disadvantages of this approach. It will be useful to compare how the approach worked for NCCS and CML_ν.

One of the biggest contrasts between NCCS and CML_ν is that of the relationship between the evaluation relation and the transition relation within each calculus. In both NCCS and CML_ν we would expect the evaluation relation to correspond to a sequence of silent actions followed by a visible action. The relationship between the evaluation relation and the transition relation for NCCS (and τNCCS) is, however, not as good a match as we might have hoped for. Considering the situation, we should not expect a perfect match. This is in the main due to the translations between NCCS and τNCCS. If these translations are removed, for example by defining the transition relation over NCCS as is done in [PR98], then the match is much closer. Furthermore, if the structural equivalence is added to the transition relation then the match is perfect. In this case the transition relation can be normalised, in the same manner as in CML_ν. If these two modifications are made then we end up with the same situation as we have for CML_ν, apart from using a different calculus.

One problem with reasoning about evaluations is caused by the use of the

structural congruence. It is very useful in reducing the number of reduction rules required to define the operational semantics. When the derivations of reductions may be normalised, as in the case of the transition relation for CML_ν, the structural congruence does not introduce any problems. We then use the method of first proving results for derivations that are simple, and then extending the result to the general case. We use the observation that a simple derivation is structure preserving. If a derivation cannot be normalised, as is the case for the evaluation relations in both NCCS and CML_ν, then the evaluation is not necessarily structure preserving and we cannot use this two stage method.

Another issue to deal with when using an evaluation relation is that, as mentioned above, it is not structure preserving. It means that proofs are significantly more complicated than those using a transition relation. This is possibly the biggest problem to consider when deciding whether to use an evaluation relation. However this may also be overcome, since by adding more evaluation rules we may create an evaluation relation that is normalisable. This evaluation relation will, however, have many rules that are derivable from some others and the (STRUC) rule. A balance has to be obtained between the simplicity of the evaluation relation, and the added power of using an evaluation relation that may be normalised.

Another significant difference between NCCS and CML_ν is the simplifications that may be applied to Evaluation Bisimulation. The results of theorem 3.7(2,3) are very useful. They mean that we only require certain contexts to determine whether two processes are equivalent, and also whether we may swap them in any context. The situation with CML_ν is considerably different. We should not be surprised that there is no such simple restriction to the context required to determine equivalence. The reasons for requiring a context are twofold, however the NCCS version only requires the first of these which is that the use of a fresh name allows a "silent action" to occur. This requires the equivalent process/program to reduce, perhaps using "silent actions", to remain equivalent. The second reason is only required for CML_ν. In CML_ν a function with a restricted channel name may be sent over an unrestricted channel. For example, consider the

program

$$P = \nu m.(m!\mathbf{true}|n!\mathbf{fn}\,(x \Rightarrow (m? \Rightarrow P')))$$

where x is not a free variable in Q. P can transmit $\mathbf{fn}\,(x \Rightarrow (m? \Rightarrow P'))$ on channel n, with the name m being restricted. We have already shown that it is sufficient to prove that for any Q, P and Q must be able to behave in the same way in any parallel context for them to be able to behave in the same way in any context. However it is an open question how much these parallel contexts may be further restricted. It is suspected that there is no simple set significantly smaller than any valid CML_ν program, or even a set merely indexed by the free names of P and Q.

A difference that may be observed in the Howe's method proofs is that while the NCCS proof used the evaluation relation, the CML_ν proof used the transition relation. This is not because the proof could not be derived using the evaluation relation, but merely that it allowed the proof to be split into smaller portions. It was felt that such an approach would be easier to understand, but should not be taken to mean that the proof was only possible using the transition relation.

We have described various difficulties that may occur using an evaluation relation rather than a transition relation. However in each case we have given methods to work around these problems. We have defined various equivalences for NCCS and one of the possible versions of Evaluation Bisimulation for CML_ν. There are a number of different decisions which may have to be taken about exactly what we allow to be observable. There are indeed many other related equivalences that we might have considered. One of these could be to have decided that if an unrestricted value is transmitted on a channel then we should be able to examine the value on it's own and not just in the presence of the transmitting expression, as is the case for the equivalences given in [FHJ95]. However this does not work. Consider the value $\text{fix}\,(x = \text{fn}y \Rightarrow P)$, with n not free in P. This should be equivalent to the value $\nu n.\text{fix}\,(x = \text{fn}y \Rightarrow ((\nu m.m!n)|P))$. It will not be if we are only allowed to examine unrestricted values. However examining restricted values doesn't necessarily make sense as shown above. Therefore adopting the methodology used in this thesis is seen to be the most natural approach.

It is the thesis of this dissertation that evaluation based methods form a useful tool for reasoning about calculi for concurrent, communicating processes. It has also been shown that delay-like equivalences can be used, and form natural bisimulations for evaluation based calculi. The method of using a parallel context to give an equivalence that is a congruence relation has been demonstrated and appears to be a powerful tool for designing equivalences of this character.

Bibliography

[ACS96a] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[ACS96b] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. Technical Report RR-2913, INRIA-Sophia Antipolis, 1996.

[AdBKR92] P. America, J. de Bakker, J. N. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. In J. W. de Bakker and J. J. M. Rutten, editors, *Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group*, pages 218–205. World Scientific, Singapore, 1992.

[Ama95] Roberto M. Amadio. Translating core Facile. Technical Report TR-3-94, ECRC, 1995.

[AP93] P. Achten and R. Plasmeijer. The beauty and the beast. Technical Report 93-03, University of Nijmegen, March 1993.

[AP94] P. Achten and R. Plasmeijer. A framework for deterministically interleaved interactive programs in the functional programming language Clean. In Jaarbeurs Utrecht, editor, *Proc. of Computer Science in the Netherlands (CSN '94)*, pages 1–30. Stichting Mathematisch Centrum, 1994.

[AP95a] P. Achten and R. Plasmeijer. Concurrent interactive processes in a pure functional language. In van Vliet Utrecht, editor, *Proc.*

of Computer Science in the Netherlands (CSN '95), pages 10–21. Stichting Mathematisch Centrun, 1995.

[AP95b] P. Achten and R. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Proc. POPL 92*, 1992.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapporte de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[FH95] W. Ferreira and M. Hennessy. Towards a semantic theory of Concurrent ML. Technical Report 95:02, School of Cognitive and Computing Science, University of Sussex, 1995.

[FHJ95] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. Technical Report 05/95, Computer Science, University of Sussex, September 1995.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A symmetric integration of concurrent and functional programming. In *Proc. Tapsoft 89*, volume 352 of *LNCS*, pages 184–209. Springer-Verlag, 1989.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Gor95] Andrew D. Gordon. Bisimilarity as a theory of functional programming. In *Proc. MFPS 95*, number 1 in Electronic Notes in Comp. Sci. Springer-Verlag, 1995.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hen94] M. Hennessy. A fully abstract denotational model for higher-order processes. *Information and Computation*, 122(1):55–95, 1994.

[Hen96] M. Hennessy. A fully abstract denotational semantics for the π-calculus. Technical Report 96:04, University of Sussex, 1996.

[HL92] M. Hennessy and H. Lin. Sybolic bisimulations. Technical Report 92:01, University of Sussex, 1992.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[How89] D. J. Howe. Equality in lazy computation systems. In *4th Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society Press, Washington, 1989.

[How96] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, February 1996.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. Object-based concurrent computing. In *Lecture Notes in Computer Science*, volume 612, pages 21–51. Springer-Verlag, 1992.

[Jef98] Alan Jeffrey. Semantics for core concurrent ml using computation types. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 55–89. Cambridge University Press, 1998.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 96)*, pages 295–308. ACM Press, 1996.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[Mil85] R. Milner. Lectures on a calculus for communicating systems. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 197–221, Berlin, 1985. Springer-Verlag. Lecture Notes in Computer Science Vol. 197.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90] R. Milner. Functions as processes. In *Proceedings of ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1990.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proc. 19th Int. Coll. on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, Berlin, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NN95] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. DAIMI-PB 483, 1995.

[PR98] A. M. Pitts and J. R. X. Ross. Process calculus based upon evaluation to committed form. *Theoretical Computer Science,*

195:155–182, 1998. A preliminary version of this paper appeared in CONCUR'96, Lecture Notes in Computer Science Vol. 1119 (Springer-Verlag, Berlin, 1996), pp 18–33.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1998.

[Rep92] J. H. Reppy. *Higher-order Concurrency.* PhD thesis, Department of Computer Science, Cornell University, January 1992.

[Rut92] J. J. M. Rutten. Semantic correctness for a parallel object-oriented language. In J. W. de Bakker and J. J. M. Rutten, editors, *Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group*, pages 272–314. World Scientific, Singapore, 1992.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[San93a] D. Sangiorgi. From pi-calculus to higher-order pi-calculus—and back. In *TAPSOFT.* Springer Verlag, LNCS 668, 1993.

[San93b] D. Sangiorgi. An investigation into functions as processes. In *Proc. Ninth International Conference on the Mathematical*

Foundations of Programming Semantics (MFPS'93), volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer Verlag, 1993.

[San94] D. Sangiorgi. Bisimulation in higher-order calculi. In *Proc. IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 207–224. North-Holland, 1994.

[San95] D. Sangiorgi. Bisimulation in higher-order calculi. Technical Report RR-2508, INRIA-Sophia Antipolis, 1995. To appear in *Information and Computation.* Revised version of the homonym paper appeared in the proc. IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET'94), North Holland, 1994.

[SS76] G.L. Steele and G.K. Sussman. Lambda, the ultimate imperative. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, March 1976.

[Sta95] I. D. B. Stark. Names and higher-order functions. Technical Report 363, Cambridge Univ. Computer Laboratory, April 1995.

[Tar55] A. Tarski. A lattice-theoretical fix-point theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tho93] B. Thomsen. Plain CHOCS. A second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, 1993.

[TLP+93] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz Knabbe, and Alessandro Giacalone. Facile antigua release programming guide. Technical Report 93-20, ECRC, 1993.

[Wal92] David Walker. Objects in the π-calculus. Technical Report CS-RR-218, University of Warwick, Department of Computer Science, April 1992.

[Wei89] W. P. Weijland. *Synchrony and Asynchrony in Process Algebra.* PhD thesis, Univ. Amsterdam, 1989.