Abstract

Communication is an integral part of computing; the last two decades have seen a merger of the fields of computer science and data communication. There is a similar trend in mathematical models of computation; recent formalisms such as the π-calculus merge the notions of computation and communication. In both theory and practice the concept of a name is used to represent references to resources and services. Dynamic binding of a name means that it refers to different things in different contexts. This lies at the heart of modular, incremental, and object-oriented programming; it is especially relevant for mobile code where a program can migrate to different sites and names represent local resources. But formalisms such as the π-calculus almost exclusively use another binding mechanism, static binding, where the referent of a name is uniquely determined by the surrounding code.

This thesis is about the nature of dynamic binding of names in algebraic formalisms for mobile systems. We add dynamic binding as an operator called blocking to the first- and higher-order π-calculi, and explore the consequences for the algebras. In particular, in the presence of blocking, Sangiorgi’s methods for reducing higher-order communication to first-order breaks down. We further investigate how notations based on the π-calculus, notably the recent λπ-calculus by Hennessy and Yoshida, may be extended with a refined notion of blocking and its dual filtering. These operators are applied to cases involving resource access sharing, access rights, and protection domains. A detailed comparison is made with the static type system proposed by Hennessy and Yoshida.

We introduce a notation based on the reflexive chemical abstract machine, extended with primitives akin to blocking to support contexts with dynamic binding. We show how this notation is suited to encode object-based systems, and present a large example involving reflection, objects which update their own code, based on a fragment of Smalltalk-80. Finally, we examine the nature of naming and name binding by investigating how these phenomena appear in existing systems and formalisms. Although static binding is still prevalent for historical reasons, our conclusion is that dynamic binding is a fundamental mechanism in mobile systems.
A mis padres
y a mamá Laura
Most papers in Computer Science describe how their author learned what someone else already knew.

Peter Landin, ca 1967
Acknowledgements

First of all I wish to thank my main advisor Prof. Joachim Parrow and my previous advisor Dr. Mads Dam, who have introduced me into this area, for the supervision of this thesis and for providing the necessary support at different stages of this work.

I would like to thank also my colleagues at the Department of Microelectronics and Information Technology, and especially Gunnar Övergaard and Pablo Giambiagi, who have always been willing to share their knowledge about object-oriented programming and mobile code security.

I am grateful also to the administrative staff of the Department, to my colleagues at the Swedish Institute of Computer Science (SICS), and to the staff of the Electrum Library.

Last but not least, I have to thank my family, Helen, Amanda and Maya, for their support and forbearance under these restless years.
# Contents

1 Introduction ............................................................. 1  
  1.1 Names .......................................................... 2  
  1.2 Background and Motivation ...................................... 2  
  1.3 Dynamic binding .................................................. 4  
      1.3.1 Dynamic Binding in Programming Languages .............. 4  
      1.3.2 Dynamic Binding in Programming Environments .......... 6  
      1.3.3 Dynamic Binding in Systems of Mobile Agents .......... 6  
      1.3.4 Dynamic Binding in Process Algebraic Formalisms ....... 7  
  1.4 The λ-calculus Paradigm ......................................... 8  
  1.5 Contributions ................................................... 9  
  1.6 Overview of the Thesis .......................................... 9  

2 Calculi for Mobility .................................................. 11  
  2.1 The π-calculus ................................................... 11  
      2.1.1 Syntax .................................................... 11  
      2.1.2 Semantics ............................................... 12  
  2.2 The π₁-calculus, the π₈-calculus, and the Receptive Distributed π-calculus ......................................................... 13  
  2.3 The Distributed π-calculus ...................................... 14  
  2.4 Mobile Ambients .................................................. 14  
      2.4.1 Syntax and Mobility Primitives .......................... 15  
      2.4.2 Operational Semantics .................................... 15  
      2.4.3 Communication ........................................... 16  
  2.5 The Reflexive CHAM and the Join Calculus ..................... 17  
      2.5.1 Syntax .................................................... 17  
      2.5.2 Operational Semantics .................................... 19  
      2.5.3 The Distributed Join Calculus ........................... 19  
      2.5.4 Binding in the Join Calculus ............................ 20  
  2.6 LLinda - Locality Based Linda ................................... 22  
  2.7 The Seal Calculus ............................................... 23  
  2.8 Other Related Calculi and Systems ............................. 24  

3 Dynamic Binding in the π-calculus .................................. 26  
  3.1 Introduction ..................................................... 26  
      3.1.1 Cryptographic Protocols .................................. 27  
      3.1.2 Modeling Cryptographic Protocols in the Higher-Order π-calculus ......................................................... 28  
      3.1.3 Extension of the π-calculus with Blocking ............... 29  
      3.1.4 Overview of the Chapter ................................... 29  
  3.2 The First-Order π-calculus with Blocking ....................... 30  
      3.2.1 Agents and Transitional Semantics ....................... 30  
      3.2.2 Strong Bisimilarity ....................................... 32
# 6 Naming and Dynamic Binding

6.1 The Semantics of Names ........................................... 134
  6.1.1 Pure Names .................................................. 134
  6.1.2 Nominal Calculi ............................................. 135
  6.1.3 The Properties of Names ..................................... 136
  6.1.4 Milner’s Object Paradigm .................................... 138

6.2 The Functional Paradigm ........................................ 140
  6.2.1 Recursion and Self-Reference ................................. 141
  6.2.2 Environments and Closures .................................... 144
  6.2.3 Denotational Semantics of Objects as Closures ............ 149
  6.2.4 Denotational Semantics of POOL ............................ 155
  6.2.5 The λ-calculus with Names ................................... 159
  6.2.6 Survey of relevant languages and formalisms ............. 161

6.3 The Object Paradigm ............................................. 174
  6.3.1 Transducers .................................................. 175
  6.3.2 Actors ....................................................... 179
  6.3.3 Modelling Objects in Process Algebraic Formalisms ....... 180

6.4 Conclusions ...................................................... 193

# 7 Conclusions ..................................................... 194

References .......................................................... 197

A Appendix .......................................................... 210
  A.1 Appendix to Section 3.2 ....................................... 210
  A.2 Appendix to section 3.3 ....................................... 221
Chapter 1

Introduction

The evolution of computer technology since the 1950s appears as a progressive transition from closed, self-contained systems composed of independent entities that do not communicate with each other, to systems where communication plays an increasingly prominent role. In the early days of computing, programmers would have the computer all to themselves while they were using it, typically with the help of an operator, to run each job in isolation. Later, so-called time-sharing enabled several programmers to share a computer transparently, as if each one were alone on it. The next stage was the introduction of distributed systems, in which no distinctions were made between local and remote operations. Workstations were connected in a network allowing remote access of printers and memory; a program invoked in a workstation could be run in another, and file systems could be shared by all users. Distribution was assumed to be transparent to the user, and the system appeared virtually as one local machine. The most recent step in this development is characterised by networking and internetworking, where a user is supposed to be aware of the existence of several machines located at different places and in different networks. Those networks can be classified according to their geographical extent, and several networks may be connected to each other in what is also called an internet. We refer to those systems as open systems. The largest internet in the world today is widely known as the Internet, with capital I.

This development is characterised by the increasing importance of data communication, which may be described as the process of moving data electronically from one point to another. Currently we see a merger of the fields of computer science and data communications, which in practice have become indistinguishable from each other. There is in principle no fundamental difference between data processing and data communication, or between computation and communication.

A similar trend can be observed in the evolution of mathematical models of computation. In the early days, these models dealt exclusively with sequential computation where notions of communication were absent. Currently we have computational models that are based entirely upon the notion of communication, and where computation and communication are equivalent terms. In this thesis we show that some aspects of earlier models, related to the way that resources are referenced and accessed, still permeate newer models, although in the context of open systems other schemes would be better motivated. A more appropriate technique will be suggested in this work.
1.1 Names

Common to both data communications and the related mathematical models are the fundamental notions of name and naming. In order to communicate with something, one must be able to refer to it, and the basic way of referencing something is by naming it. However, names also occur in messages. Communicating something is naming something, e.g. a resource, an object, a piece of code, or a service.

As an example, a handler that receives a service request must interpret the name of the required service, and can delegate the request to any other handler it finds suitable for the occasion. A client needs not be aware of how a service is implemented, or how and where it runs. This way of binding a name to some local entity during runtime is the essence of what is known as dynamic binding. It enhances flexibility and extensibility by allowing various software components to be linked dynamically. The mechanism of name binding lies also at the heart of modular, incremental, and object-oriented programming. It allows a name to be bound to distinct entities, created at different occasions, and possibly on separate locations.

Names and dynamic binding are ubiquitous in computing. In imperative languages, the notion of state is defined as a function from names to values. Unless a name is defined as a constant, its value may change during computation, and must be resolved at runtime whenever it is accessed. In operating systems names are used to access commands and files. Dynamic binding of names also appear when modules are loaded or shared libraries searched at runtime. Applets or WWW browsers in Java are dynamically loaded through links occurring in a WWW page. Objects are often formalised as records consisting of an association of names to expressions, and incremental programming environments use names to keep track of program development. In multiprogramming, indirect addressing in dynamic storage allocation requires an association mechanism involving names and locations. In relational database management systems names are used to identify and organise databases, relations and attributes. Distributed systems and networks use names to support sharing of resources and communication of information.

This thesis is about the nature of dynamic binding in open systems. In spite of its significance, dynamic binding still remains to a large extent semantically obscure, and the concepts of naming and dynamic binding of names have not received an adequate and systematic formal treatment to this day. We intend to explore the notion of dynamic binding in the context of what are known as process algebraic formalisms. We show how dynamic binding of names may be used to encode the functionality exhibited by open systems, to reflect the structure induced by the linkage among their components, and to capture the idea of location and local control.

A central claim of the thesis is that the notions of naming and dynamic binding of names are essential in systems of communicating agents, and consequently in any formalism intended to encode them. In addition, we consider the security-related notions of protection domains and access rights, which turn out to be closely related to our subject.

1.2 Background and Motivation

In many systems the meaning of names can be statically defined and given a unique and unchanging reference. Many current models of computation and formalisms for distributed systems were created for describing the behaviour of local area networks connecting autonomous computers for the purpose of resource sharing. This kind
of system typically consists of a collection of relatively uniform and static environments containing nodes and links between nodes. They are characterised by their predictability with respect to bandwidth fluctuations and process failures, and by the availability of a global address space, a global file system, and a centralised administration ensuring reliability, stability, and a high level of protection. Mobility in a variety of forms is present in those systems, albeit in a context where issues concerning locality of computation can be abstracted away. This framework encouraged the adoption of a static binding discipline in many programming languages and program formalisms, to the detriment of dynamic binding.

Nevertheless, these assumptions do not hold in the context of computational structures such as the World-Wide-Web and wide-area networks. Access to resources is unreliable and opaque in such networks; there is no support for a global file system, no integrity of addressing, no single reliable components or failure points, and no centralised administration. In contrast to local area networks, those systems can be best described as a consisting of a heterogeneous, mutually distrustful and dynamic collection of computers and independent administrative domains provided with changing links and a decentralised administration. Systems of this kind are intrinsically unreliable, unpredictable, and unmanageable as a whole.

With the advent of the WWW, the phenomenon of mobile code that migrates from site to site in quest of new resources is becoming very common. Typically, a distant site holds a resource that is not available at the site where the computation is currently running. The computation may thus migrate to the site holding the resource and access it locally. A name that is intended to be bound dynamically in this setting is *hostname*, which should sensibly refer to its latest active binding.

Mobile computation is thus becoming increasingly common in a context where issues of locality cannot be abstracted away. Security concerns require that access barriers cannot be made transparent and must be treated as a fundamental feature of distributed computation. Any computational model for this kind of system must be endowed with primitives that are suitable for structuring the computational space around notions of hierarchic domains, barriers, and mobility. Dynamic binding of names is imperative in this context.

Object-based programming methodology is being increasingly regarded as a good basis for the design and implementation of modern open systems. Objects have been shown to be useful as an organisational paradigm for decomposing large systems into smaller and more manageable ones, and for code reuse.

An open object-based system is a dynamic heterogeneous collection of loosely interconnected nodes, supporting applications defined as persistent objects with an internal state and an external interface for interaction with the environment. Objects execute concurrently and are capable of communicating asynchronously with each other via message passing. Internally, an object may consist of several encapsulated objects that are not visible from the exterior. Local state is encapsulated to ensure protection against attempts to change it by actions at distance. Objects may change state as a result of service requests.

The notion of name is central in open object-based distributed systems. Public or shared names are a pre-condition for the establishment of any form of communication among distinct parties. External coordination of objects for the establishment and execution of a dialogue among several parties cannot take place in the absence of conventions or protocols about the use of names as identifiers of objects, addresses, resources or services. For internal coordination encapsulation primitives are needed to hide the scope of names in order to prevent interference from the outside.
Naming in open systems must be flexible. It should be possible to create fresh user-defined names, and also to hide the scope of names within an environment by encapsulation. Global computing renders global name spaces unrealistic. As a result, encapsulation mechanisms are necessary to support the definition of contexts or environments for local binding of names. Therefore, both static and dynamic binding of names might be called for.

1.3 Dynamic binding

A binding consists of an association of a name to some kind of object entity. Examples of binding pairs are a service and a server, a method name and a sequence of code, an object name and an object, or a program variable and a memory address. Names may also refer to the value of the individual variables of a program. An environment can be defined as a finite set of bindings; moreover, a collection or hierarchy of environments can also be called an environment or a context. A name is resolved within an environment by looking up a binding pair where the first element is the name itself.

In programming languages, static binding, also known as early binding, refers to a name resolution performed at compilation time by the compiler, which may eliminate names by turning them into machine addresses. In dynamic binding, by contrast, names cannot be eliminated since they must be resolved at runtime, typically by being looked up in tables representing the environment.

Theoretically, dynamic binding is often regarded as less attractive than static binding, although the definition of the semantics of programming constructs may occasionally be simpler for the former. Expressions containing unbound variables are not referentially transparent, and must in consequence be represented by their unevaluated text. Moreover, some modularity issues seem to talk against dynamic binding, due to possible symbol conflicts. In the context of mobile computing, similar reasons have been put forth in favour of static binding. This discipline supports a consistent semantics of distributed computation, since program constructs can be given a precise meaning by the immediate text surrounding them, thereby simplifying the task of reasoning about the program. But as we shall see in this section, there are other reasons that speak in favour of dynamic binding.

1.3.1 Dynamic Binding in Programming Languages

Functional and imperative languages

By way of illustration, suppose we have the following definitions in a Lisp-like syntax:

```
(define (n 2))
(define (g x y) (x + y))
(define (f x) (g(x,n)))
(define (h F n) (F(n)))
```

For a language with static binding, the value of the expression \(h(f,1)\) is 3. However, a language with dynamic binding will evaluate the expression to 2. The first step of this evaluation yields the expression \(f(1)\) and an environment where \(n\) is bound to 1. The expression \(f(1)\) is then evaluated to \(g(1,n)\). The latter expression is finally evaluated in the environment returned by the evaluation of the expression \(h(f,1)\).

Observe that this scheme would not work if the name \(n\) is substituted by some other name in the definition of \(f\) or \(g\). Dynamic binding turned out to be helpful in structuring large programs, and may be viewed as a form of communication between
distinct programs, or distinct parts of a program developed within an incremental programming environment.

However, the most interesting cases concern the binding of a function body to a function call. Thus, given an assignment of type

\[ a = f(b) \]

we have to ask when the decision will be made as to what code will be bound to the function identifier \( f \). Traditional compiled imperative languages make this decision at compile-time. The resulting binding is immutable, i.e. the same code will always be invoked whenever \( f \) is evaluated. In contrast, interpreted languages such as LISP makes the decision at runtime.

It was early found that dynamically bound variables can be helpful in structuring large programs by simplifying procedure calls, and also that dynamic binding simplifies the implementation of languages with functions as first class values [162, 6]. This is basically why McCarthy’s Lisp, and other interpreted languages, adopted the dynamic binding discipline. Dynamic binding is often associated with Lisp and later implementations such as MacLisp [100] and Gnu Emacs Lisp [88]. However, the earliest versions of LISP had no well-defined notion of a free variable. In fact, dynamic binding appeared first as a bug, and its solution led to the implementation for the first time of closures (by Steve Russell [101]).

Thus, although static binding has become the norm in functional languages, dynamic binding remains an expressive and powerful programming technique, and many languages based on static binding also provide it in some form, e.g. CommonLisp [159] and MITScheme [62]. Other languages that support dynamic binding are Perl [182], TEX [86] and Bash [141]. Furthermore, Moreau [116] proved that dynamic binding in fact adds expressiveness to a purely functional programming language, according to a definition of observational equivalence that uses the evaluation function.

**Object-oriented programming languages**

In object-oriented programming languages (OOPL), *methods* associated with objects or classes, can be invoked via message passing, as in the expression

\[ o.m(b) \]

where \( o \) denotes an object’s identifier or a reference to an object, \( m \) a method defined in the class of \( o \), and \( b \) the actual parameter. Objects are organised according to an inheritance or subclass hierarchy; since the referent of an object identifier \( o \) may not be known at compile-time, the choice of a method code is done at runtime. Nevertheless, even if the referent of \( o \) is known at compile-time, two different invocations of the method \( m \) involving two objects \( o \) and \( o' \), e.g. in

\[ o.m(b); o'.m(b') \]

may result in the execution of two different codes, for instance if the classes of \( o \) and \( o' \) are distinct. We can here regard the method denoted by \( m \) as a polymorphic function parameterised on the type or class of the object \( o \).

It is in the field of object-oriented programming languages that the notion of dynamic binding, also known as late binding or dynamic method lookup, plays a most prominent role. In Smalltalk, for instance, all function calls are resolved by dynamic binding, and in C++ we have both options, compile-time or dynamic binding. The
mechanism that implements dynamic binding in OOPL is called dynamic dispatch, an operation in which runtime information is used to look up the code of a method and execute it.

Dynamic binding is one of the most important semantic aspects of OOPL. Nevertheless, in the process algebra formalisms that have so far been proposed to encode OOPL constructs, dynamic binding is conspicuously absent, a fact which has originated unnecessarily contrived encodings of objects.

### 1.3.2 Dynamic Binding in Programming Environments

In traditional sequential programming languages without dynamic binding and without modularity, names can be eliminated by being turned into memory addresses by the compiler. As a result, no symbol tables, and consequently no lookup operations, are required during execution of the program. The names occurring in the source code of the program may thus be regarded as mere place-holders for hardware references, and can in consequence be substituted in the source code by other names. This resembles the role played by variables in the $\lambda$-calculus, which may be rightly characterised as a name-free calculus.

However, whenever we have to do with a collection of independently constructed programs, intended as subcomponents of a larger system, with perhaps extensible user-defined libraries, names start playing a more fundamental role, and renaming might change the meaning of a program. The reason is that names are now used as a means of coordination among distinct parts of programs. Programs must be able to call on other programs and refer to data objects defined elsewhere by name. Modularity requires that an object may be referenced via its interface, which usually consists of a set of names. Hence, as soon as some form of communication between independent entities of a system is required, symbol tables binding names to other values become an indispensable part of the runtime environment of the system.

### 1.3.3 Dynamic Binding in Systems of Mobile Agents

Dynamic binding is also present in systems of mobile agents, where object structuring concepts have played an important role. A mobile client may be bound to a service rather than to a server, enabling the client to switch servers dynamically. Mobile applications that depend on information that is local to a server require dynamic binding of names.

Dynamic binding in mobile programs might enhance both security and flexibility. A mobile program must be able to name site-specific operations in order to access local resources at the target host, and this mapping between names and operations must be established dynamically. This also enables the target host to retain a certain control over the execution of mobile programs.

Since open systems are not under the control of a single administrator, they cannot rely exclusively on static configuration for their maintenance. It is unrealistic to assume a homogeneous reference system with universally unique identifiers in large-scale systems such as the Internet. The computational structure of the Internet violates many assumptions about the behaviour of distributed systems in local-area network architectures. For instance, in the Internet we have to face the existence of distrustful administrative domains erecting barriers or firewalls against each other in order to hide or impede access to local resources.

In a context in which security concerns must be taken into account, the functionality provided by dynamic binding might be essential. In order to enhance
security, the meaning of many program constructs should be often determined by the context of the execution site. This is especially important when higher order constructs, such as mobile agents, are present. In this framework, a server may accept network-transmitted procedures for execution containing occurrences of free, unbound identifiers, that should acquire a meaning determined by the receiving host.

1.3.4 Dynamic Binding in Process Algebraic Formalisms

Process calculi, or calculi for mobile agents, are a large class of formal systems created with the aim of being used as a formal basis for concurrent and distributed programming languages and systems, in analogy to the way that the $\lambda$-calculus is used as a formal basis for functional computation.

The importance of dynamic binding in formalisms for mobile processes is now being increasingly recognised. The presence of higher-order communication, i.e. communication of processes, in programming language and formalisms seems to bring to a sharper light many aspects related to dynamic binding. In process calculi, one of the earliest attempts to incorporate higher-order communication in the notation is Thomsen’s CHOCS [168], an extension of Milner’s CCS [109]. A different version of CHOCS, called Plain CHOCS, was also formulated, in which dynamic binding is eliminated by being turned into a static scope binder.

The $\pi$-calculus introduced by Milner, Parrow and Walker [112] contains only first-order features. In this calculus, name passing simulates process passing, and name restriction, a static scope binding operator, acquires a very important role. Dynamic restriction in the style of CHOCS is absent. Thomsen [168] compares this scheme to the higher-order approach adopted in CHOCS, but only with respect to Plain CHOCS. Transmission of a process in CHOCS is translated to the $\pi$-calculus as transmission of a name acting as a link to a trigger construct providing copies of the process. This resembles invocation of procedures in programming languages with static binding of variable names. Thomsen declared that the translation ensured static binding since the process stayed in the sending environment. However, we could turn this around and say that the static binding discipline ensured the translation, since the scheme of sending links to triggers of processes, instead of the processes themselves, breaks down in the presence of dynamic binding. Apparently, the issue of the reduction of CHOCS with dynamic restriction to the $\pi$-calculus has not been considered further, although some of the features of dynamic restriction are attractive, for instance the fact that dynamic binding yields a simpler operational semantics in CHOCS. Another interesting aspect of dynamic binding in CHOCS is the fact that it enables the definition of a recursion construct similar to the paradoxical combinator in the $\lambda$-calculus. In Plain CHOCS the possibility of defining a similar construct was only conjectured by Thomsen. However, it was shown later that the first-order calculus is operationally complete, and that the higher-order variants of this calculus could be faithfully reduced to the first-order $\pi$-calculus (Sangiorgi [149]). As a result, the matter seemed to be settled in favour of the first-order paradigm. Higher order constructs were regarded as convenient for reasoning at a more abstract level, but nothing more. In this thesis we intend to reconsider this fact by arguing that in the presence of dynamic binding, higher order process passing enhances expressiveness, and may no longer be reduced to name passing in a simple way. This is the subject of Chapter 3. Moreover, the almost universal presence of dynamic binding in mobile systems seems to have affected many recent process algebraic formalisms, a fact that bears evidence of the central claim of this thesis.
1.4 The $\lambda$-calculus Paradigm

The role played by the $\lambda$-calculus as a basis for programming language semantics can be described as paradigmatic. The semantics of pure functional programming languages usually assumes the existence of data structures upon which transformations are performed by a computational agent according to a well-determined set of rules. The data structures can be represented by symbolic expressions composed of variables. In fact, variables are not essential to the calculus, as attested by the fact that it can be reduced to a purely positional system, cf. de Bruijn indices, or to a combinatory calculus requiring only a small number of constants called combinators. In either case the notion of name as we understand it here is absent. The variables occurring in any expression can be substituted by other variables without changing the semantics of the expression, as long as the substitution does not identify distinct symbols.

This notion of computation usually extends to concurrent systems. As a result, many researchers strive towards giving a semantics to those systems in terms of name-free formalisms. This is evidenced by the fact that there has been an overwhelming preference for the static binding discipline. The alleged reason is that static binding yields a more tractable semantics, which might be a consequence of the fact that dynamic binding is absent from the $\lambda$-calculus, the basis for those semantics. However, in systems involving communicating agents the dynamic binding discipline is essential. Therefore, although static binding might arguably yield a more tractable semantics, its adoption hereby to the detriment of dynamic binding seems like looking for lost keys where the light is better.

In process algebraic formalisms we have also seen the development of a semantics based on the notion of unlabelled transition systems, in the spirit of the $\lambda$-calculus, thus eliminating occurrences of names in the labels of a labelled transition semantics. In fact, the significance of names in process algebra, largely restricted to their role as channel names or as unique identifiers of objects, has been recognised ex post facto and only fragmentarily.

The $\lambda$-calculus paradigm also lies behind what seems to be the only alternative to communication by naming, namely communication by position, which in effect is a kind of name-free communication. In the $\lambda$-calculus, application of an abstraction may be seen as a form of communication where the formal parameters become bound to the actual parameters. The main problem with this solution is that it requires that communication parties follow a very tight protocol. In traditional object-oriented programming this method implies that the sender of a message must have information about the specific class to which the receiver object belongs, since by subclassing an object might have an interface that is larger than the interface of its superclass. We will return to this issue in Section 6.2.

Nevertheless, computing has also been viewed in terms of symbol manipulation [125], a line of thought which probably inspired Milner [111] to make what he called the ontological commitment of seizing upon the notion of naming as the central one in concurrent computation. However, this basic insight has apparently not been very influential, probably because researchers in the area are more inclined to work with concepts that are natural to formal logics and the $\lambda$-calculus paradigm, and naming is certainly not one of them.
1.5 Contributions

Several researchers have pointed out the limitations of the functional approach in systems where the notion of state is central, as in object-based systems. We believe that the interactional aspects of programming are best expressed in an object-based framework, where entities called objects have state that evolves by interaction with other objects. However, there has been a clear tendency to study objects within the framework of the functional paradigm. It has been argued that the semantics of programming languages requires well-defined mathematical values that are to be found in the denotational and functional approach.

The object paradigm introduced by Milner constitutes a more adequate approach in this context. Milner contrasts the functional paradigm with the “links-as-values” approach, and identifies the “act of naming” as the central notion in models of interaction. What we propose in this thesis is the adoption of a more general notion of naming within the framework of the object paradigm.

The main contributions of this thesis are the following:

- We show an extension of both the first-order and the higher-order \( \pi \)-calculus with a new operator for encoding the notion of dynamic binding, called blocking, and the establishment of several important algebraic, expressivity and equivalence properties (Chapter 3). It is shown that the extension of the first order \( \pi \)-calculus with the blocking operator yields a tractable algebra and preserves the most important properties from the basic calculus. Results are also shown concerning the equivalence in expressiveness between the blocking operator on the one hand, and the matching and mismatching operators on the other. Finally, a reduction from the higher order calculus with blocking to the first order one is presented, as well as a detailed proof of the adequacy of the translation.

- We introduce the notion of polarised channels in the \( \pi \)-calculus with blocking, as well as a new operator, the filter operator, for controlling the communication capabilities of a process and for modelling access rights. We add the filter and the blocking operator to a typed higher-order version of the \( \pi \)-calculus, the \( \lambda \pi_c \) calculus. A simplification of the type system of the latter is presented, as well as an encoding of the notion of process interface of the \( \lambda \pi_c \) calculus in terms of the filter operator. A proof of the adequacy of the encoding is given (Chapter 4).

- We develop a new notation inspired by the Reflective Chemical Abstract Machine and the Join Calculus, endowed with primitives for encoding the notion of context or environment. The notation is used to encode common data structures. We also formulate a higher-order version of the notation, intended to encode the notion of reflection in a system of objects with a hierarchic structure similar to the one found in Smalltalk, and give an encoding of a large example based on this language (Chapter 5).

- We present an extensive study on the development of the notions of naming and dynamic binding in programming languages, systems and formalisms (Chapter 6).

1.6 Overview of the Thesis

The rest of this thesis is organised as follows. In Chapter 2 we introduce briefly some of the main formalisms proposed for mobile processes. Chapter 3 is dedicated
to a presentation of the results obtained by extending both the first-order and the higher-order π-calculi with the blocking operator. A couple of reductions are also shown, the first one from the first-order π-calculus with blocking to the first-order π-calculus with mismatching, and the second one from the higher-order π-calculus with blocking to the first-order π-calculus. The correctness of these reductions is proved in two separate appendices. In Chapter 4 we refine the notion of blocking in the first-order π-calculus by introducing polarities. We also introduce the filter operator, and compare the approach for encoding access rights and capabilities based on typing with an approach which we propose here and that is based on blocking and filtering. For this purpose, modifications are introduced to a higher-order typed version of the π-calculus in order to compare both approaches. In Chapter 5 we formulate a new formalism inspired by the Reflexive Chemical Abstract Machine and the join calculus, which we call Context Reflexive Chemical Abstract Machine, briefly CRCHAM, intended to capture the notions of locality and late binding in object-based systems. Several short examples are given, as well as an extensive one where we present an encoding of the reflective class structure of a fragment of Smalltalk-80. In Chapter 6 we focus on the issues of naming, dynamic binding, environments, and the functional and object paradigms, the latter also called the concurrent paradigm. Previous work in the field is outlined and discussed. In Chapter 7 we present some conclusions and suggestions for further work.
Chapter 2

Calculi for Mobility

In this chapter we present some of the most relevant process algebraic calculi and formalisms intended for modelling mobile systems. We discuss also some aspects of these formalisms that are specially relevant for this thesis. We concentrate here on notations that have a general-purpose nature. In Chapter 6 we present other notations intended for more specialised purposes.

2.1 The $\pi$-calculus

The $\pi$-calculus is an algebraic process formalism that is based on the notion of name [112]. It describes networks that can configure themselves, and provides a model of communicating behaviour that treats mobility in terms of message transmission. Its theory includes the important concept of observational or behavioural equivalence of processes.

2.1.1 Syntax

An infinite set of names $\mathcal{N}$ is assumed, as well as set of agent identifiers where each agent identifier $A$ has a nonnegative arity. We use $x, y, z, w, v, u$ as metavariables over names.

**Definition: Agents** The agents of $\pi B$ are generated according to the following abstract syntax, using $P, Q, R$ as metavariables over agents:

\[
\begin{align*}
P & := \ 0 \quad \text{(inaction)} \\
& \quad | \pi y. P \quad \text{(output prefix)} \\
& \quad | x(y). P \quad \text{(input prefix)} \\
& \quad | \tau. P \quad \text{(silent action)} \\
& \quad | (x)P \quad \text{(restriction)} \\
& \quad | [x = y]P \quad \text{(match)} \\
& \quad | P \mid P \quad \text{(composition)} \\
& \quad | P + Q \quad \text{(summation)} \\
& \quad | A(y_1, \ldots, y_n) \quad \text{(identifier)}
\end{align*}
\]

The null agent $0$ is inactive. Prefixed agents have shape $\alpha. P$, where $\alpha$ is either an input prefix $x(y)$, an output prefix $\pi y$, or the silent action $\tau$. Prefixed agents are agents that must perform an input or output action before continuing. The prefix $\pi y$ is called a negative prefix; $\pi y. P$ may be thought as an agent that must send the
port $y$ along the output port $x$ before continuing as $P$. The prefix $x(y)$ is called a positive prefix; $x(y).P$ denotes an agent that must input an arbitrary name $z$ along the port $x$ before continuing as $P\{z/y\}$, i.e. the agent $P$ with occurrences of $y$ are substituted by $z$ in a way that avoids name capture, cf. the notion of substitution in the $\lambda$-calculus. Occurrences of $y$ in $P$ are bound by the prefix $x(y)$ in $x(y).P$.

The expression $\tau.P$ denotes an agent that must execute a silent action $\tau$ before continuing as $P$. Summation represents alternative choice, and composition allows processes to evolve concurrently. A restriction $(x)P$ denotes an agent $P$ in which occurrences of $x$ in $P$ are bound by $(x)$. A match $[x = y]P$ acts like $0$ if it is not the case that $x = y$, otherwise like $P$. The match operator is also called the matching operator in the literature. In his paper we use both terms undistinguishably. In many notations the restriction operator $(x)P$ is represented as $(ux)P$. When dealing with these notation we respect this syntax.

The free names $\text{fn}(P)$ of $P$ are those names which occur in $P$ not bound by either restriction or a positive prefix. The bound names $\text{bn}(P)$ are those names occurring in $P$ within the scope of an operator binding the name, either restriction or positive prefix. The names $n(P)$ of $P$ is defined as $\text{fn}(P) \cup \text{bn}(P)$. Agents which are alpha-convertible are identified. We assume the existence of a unique defining equation for each agent identifier $A$ with arity $n$ of the form

$$A(x_1, \ldots, x_n) \overset{\text{def}}{=} P,$$

where the $x_i$ are pairwise distinct, $\text{fn}(A(\bar{x})) \subseteq \{x_1, \ldots, x_n\}$ and $\text{bn}(A(\bar{x})) = \text{bn}(P)$, assumed to be finite, where $\bar{x} = x_1, \ldots, x_n$.

### 2.1.2 Semantics

The semantics of the $\pi$-calculus is based on a labelled transition relation. The notation $\pi(y)$ will be used here as a shorthand for $(y)[\pi(y)$. Transitions have the general shape $P \xrightarrow{\alpha} Q$, where the action $\alpha$ is either an input action of form $x(y)$, a free output action of form $\pi(y)$, a bound output of form $\pi(y)$, or the silent action $\tau$. Free output actions and the silent action are called free actions, and input and bound output actions are called bound actions. The subject of an action $\pi(y)$ or $x(y)$, briefly $s(\alpha)$, is the singleton set containing $x$, whereas the object of an action $\alpha$ is either the free object $y$ in case $\alpha$ is of form $\pi(y)$, or the bound object $y$ if $\alpha$ is of form $x(y)$ or $\pi(y)$. If $\alpha = \tau$, $s(\alpha)$ is empty. The set of bound names $\text{bn}(\alpha)$ of an action $\alpha$ is empty if $\alpha$ is a free action, otherwise it is the singleton set containing $y$ if $\alpha$ is $x(y)$ or $\pi(y)$. The set of free names $\text{fn}(\alpha)$ of $\alpha$ consists of $s(\alpha)$ and the free object, if any, of $\alpha$. The set of names of $\alpha$, briefly $n(\alpha)$, is defined as $\text{bn}(\alpha) \cup \text{fn}(\alpha)$.

The operational semantics of the $\pi$-calculus is given in the Table 1.
TABLE 1
Rules of Action

TAU-ACT: \( \tau.P \rightarrow P \)

OUTPUT-ACT: \( \exists y. P \rightarrow y.P \)

INPUT-ACT: \( x(z). P \rightarrow (w) P\{w/z} \quad w \notin \text{ln}(\{z\}.P) \)

SUM: \( P \rightarrow P' \quad Q \rightarrow Q' \)

MATCH: \( [x = x] P \rightarrow P' \)

IDE: \( P[\{y/x\}] \rightarrow P' \quad \Lambda[y] \rightarrow P' \quad \Lambda[y] \overset{\text{def}}{=} P \)

PAR: \( P \rightarrow P' \quad P|Q \rightarrow P'|Q \quad \text{bn}[\alpha] \cap \text{ln}[Q] = \emptyset \)

COM: \( P \rightarrow P' \quad Q \rightarrow Q' \quad P|Q \rightarrow P'|Q' \{y/z\} \)

CLOSE: \( P \rightarrow P' \quad Q \rightarrow Q' \quad P|Q \rightarrow (w)[P'|Q'] \)

RES: \( P \rightarrow P' \quad \{y/x\} \quad y \notin \text{ln}[\alpha] \)

OPEN: \( P \rightarrow P' \quad \{y/x\} P' \rightarrow \{y/x\} P \quad y \neq x, w \notin \text{ln}(\{y\},P) \)

Obs: Symmetric forms for operators + and \( \bot \) have been omitted

2.2 The \( \pi_1 \)-calculus, the \( \pi_\bot \)-calculus, and the Receptive Distributed \( \pi \)-calculus

The \( \pi_1 \)-calculus is an asynchronous distributed typed calculus based on a fragment of the \( \pi \)-calculus. By extending the \( \pi_1 \)-calculus with constructs for localities, migration and failure detection we obtain the \( \pi_\bot \) calculus [12]. Every channel name is associated with a unique persistent process which answers to messages addressed to that channel. The \( \pi_1 \)-calculus is shown to be expressive enough to simulate the asynchronous \( \pi \)-calculus.

The main additions to the \( \pi_1 \) calculus are the following. A process running at location \( a \) is denoted by \{p\}a. An operator \texttt{spawn(\_)} is used to create processes at remote locations. Messages can be output actions or "particles", stop of a location \( a \) (\texttt{stop(a)}), spawning of a process \( p \) at a location \( a \) (\texttt{spawn(a, p)}), or testing of a location \( a \) with return \( b_1 \) if the location is alive, otherwise \( b_2 \) (\texttt{ping(a, b_1, b_2)}). Every location is associated with a location process which handles routines, \texttt{stop}, \texttt{spawn} and \texttt{ping} messages. This is done by the introduction of a sort \texttt{loc} and an operator denoting the creation of a location process receiving on a name \( a \) of sort \texttt{loc}, \texttt{LocT(a)}, where \( T \) is either \( R \) for \texttt{run} or \( S \) for \texttt{stop}.

The \( \pi_1 \) calculus is viewed as a way of capturing the unicity of the receptor, as in the join calculus, by means of typing rather than syntax, thus allowing for reuse of earlier results. Furthermore, it makes the communication primitives of the \( \pi \)-
calculus closer to object-oriented message sending.

The Receptive Distributed π-calculus [13] is a simplified version of the $D\pi$ calculus [64] (see Section 2.3) and an extension of the $\pi_1$ calculus that is shown to contain the latter. It is intended as a formalisation of the property of the uniqueness of name of a persistent object or process that may encapsulate a state, and to ensure the property of the unicity of receivers. In this calculus, processes are allowed to send and receive compound names $a@l$ meaning the channel $a$ at location $l$, and recursive calls $A(a; u)$, where $A$ is an identifier, meaning that the name $a$ is the only one used as an input channel in the body of the definition of $A$. The receptiveness of any migrating message is guaranteed. A process $P$ located at $l$ is denoted $[l :: P]$, and since it is the case that $[l' :: [l :: P]] \sim [l :: P]$, distributed systems are shown to have a flat structure, like the $D\pi$ calculus, in contrast to the Distributed Join Calculus (Section 2.5.3) or the Ambient Calculus (Section 2.4), where locations may be nested.

2.3 The Distributed $\pi$-calculus

A distributed variant of the $\pi$-calculus, the distributed $\pi$-calculus, shortly $D\pi$, is a typed language for mobile agents introduced by Hennessy and Riely [64]. The language allows fine control over the use of resources in a system with the help of the notion of resource access capability. An agent may be given the capability to use a channel in either input or output prefixes, but not necessarily both. Agents may move between locations and augment their set of capabilities via communication with other agents. Typing is based on the notion of location type that describes the set of resources available to an agent at a location. The language is similar to the one studied by Amadio [12] [15], but it ignores location failure, and communication is restricted to be purely local. An extension of $D\pi$ presented in [144] supports the modeling of certain forms of site failure via a “halt” primitive.

In $D\pi$ resources are channels which support binary communication between agents. Agents are described as threads, and are terms of the polyadic $\pi$-calculus extended with primitives for movement between locations as well as for creation of new locations. Locations are names, and apart from constructs denoting capabilities, the syntax of the language is similar to that of the polyadic $\pi$-calculus extended with constructs such as go $u.P$ (movement), and $k[P]$ (agent running at location $k$). Communication is local: two agents may interact if they are running at the same location. The main new reduction rule is

$$(r\text{-move}) \quad \ell[[\text{go } k.P]] \rightarrow k[P]$$

which allows an agent to move from location $\ell$ to location $k$.

In $D\pi$ mobility is associated with agents or inactive code, not locations or running code as in the Distributed Join calculus [48] or in the Ambient calculus [32]. According to the authors, this feature supports a clear distinction between resource and agent, and is well understood from previous work in concurrency theory [64]. On the other hand, location movement by itself is not enough to express interaction between agents. In the Ambient calculus this is done by the introduction of an open operation, and in the Distributed Join calculus by message movement.

2.4 Mobile Ambients

A radical departure from formalisms based on static scoping is Cardelli’s Mobile Ambients [32]. Cardelli makes the concept of location or ambient the basic primitive
of his calculus, with the purpose of modeling so-called administrative domains. In this formalism, ambients are hierarchically structured and agents are confined to ambients. The structure is dynamic: the connection between a name and an ambient is dynamic, since the navigation primitives dynamically bind a name to any ambient with the same name. Agents confined to ambients control their behaviour, and may both move or eliminate those ambients. All computation takes place within the boundaries of ambients. Each ambient has a name, used to control access, and a collection of local agents and subambients. Names may be seen also as capabilities, and may be created, passed around, or used to name new ambients.

2.4.1 Syntax and Mobility Primitives

The core of the calculus contains exclusively mobility primitives. This basic calculus is later extended to include communication primitives.

The syntax is the following:

\[ P, Q \overset{\text{def}}{=} \text{processes} \]

\[ (\nu n)P \] restriction

\[ 0 \] inactivity

\[ P \mid P \] composition

\[ !P \] replication

\[ n[P] \] ambient

\[ M.P \] action

\[ M \overset{\text{def}}{=} \text{capabilities} \]

\[ \text{in } n \] can enter n

\[ \text{out } n \] can exit n

\[ \text{open } n \] can open n

Restriction, inactivity, composition and replication are well-known operators in process calculi. Ambients are written \( n[P] \), where \( n \) is the name of the ambient, and \( P \) is a process running inside the ambient. Actions are written \( M.P \), where \( M \) is a capability and \( P \) is a process that runs after an action \( M \) has been executed. There are three kind of capabilities: \( \text{in } n \) for entering an ambient named \( n \); \( \text{out } n \) for leaving an ambient named \( n \), and \( \text{open } n \) for opening an ambient named \( n \).

2.4.2 Operational Semantics

The operational semantics of the calculus is based on the following structural congruence between processes:
Furthermore, processes are identified up to renaming of bound names:

\[(\nu m)P = (\nu m)P[n \leftarrow m] \text{ if } m \not\in \text{fn}(P)\]

The behaviour of processes is given by the following reduction rules:

\[n\text{in } mP | Q \mid mR \rightarrow m[nP | Q] | R\] (Red In)
\[m\text{out } nP | Q \mid R \rightarrow n[mP | Q] | mR\] (Red Out)
\[\text{open } nP | nQ \rightarrow P | Q\] (Red Open)
\[P \rightarrow Q \Rightarrow (\nu m)P \rightarrow (\nu m)Q\] (Red Res)
\[P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]\] (Red Amb)
\[P \rightarrow Q \Rightarrow P | R \rightarrow Q | R\] (Red Par)
\[P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'\] (Red =)

### 2.4.3 Communication

The basic Ambient Calculus presented above, consisting only of mobility primitives, is Turing-complete. However, without notions of communication or variable-binding operators it is hard to encode other formalisms. To remedy this, a simple asynchronous communication mechanism is introduced. The basic formalism is extended with input \((x)P\) and output \((\langle M \rangle)\) primitives. The following terms are added to the syntax of the pure Ambient calculus:
\( P, Q \overset{\text{def}}{=} \text{processes} \)

\[
\begin{align*}
\ldots \\
M.P & \text{action} \\
(x)P & \text{input action} \\
(M) & \text{async output action}
\end{align*}
\]

\( M \overset{\text{def}}{=} \text{capabilities} \)

\[
\begin{align*}
\ldots \\
x & \text{variable} \\
n & \text{name} \\
\varepsilon & \text{null} \\
M.M' & \text{path}
\end{align*}
\]

Capabilities are identified up to the following equations: \( L.(M.N) = (L.M).N \) and \( M.\varepsilon = M = \varepsilon.M \).

Names and capabilities may be the object of a communication. We leave out the definition of free names and free variables. It suffices to point out that \( \nu \)-bound names and input-bound variables are distinguished in the calculus. The definition of the substitution of the capability \( M \) for each free occurrence of the variable \( x \) in the process \( P \), briefly \( P \{ x \leftarrow M \} \), is straightforward and is also omitted.

The communication mechanism introduced may be described as anonymous communication local to an ambient. An output action releases a capability and an input action captures a capability and binds it to a variable within a scope:

\[(x)P | (M) \rightarrow P \{ x \leftarrow M \} \].

### 2.5 The Reflexive CHAM and the Join Calculus

The reflexive CHAM \([47]\) is obtained from CHAM \([21]\), the chemical machine of Berry and Boudol, by adding reflexion, the power to dynamically create new reaction patterns. The Join Calculus has been proved to be expressively equivalent to the \( \pi \)-calculus.

The Join Calculus is the syntactic description of the reflexive CHAM molecules. Basically, it combines restriction, reception and replication in a single receptor definition, where a join pattern provides synchronisation capabilities similar to synchronous channels. It is a name passing calculus with an emphasis on distributed programming and constitutes the core of a programming language with a distributed implementation.

#### 2.5.1 Syntax

The syntax of the reflexive CHAM assumes an infinite set of names \( \mathcal{N} \), ranged by \( u, v \). Variables in lowercase letters \( x \) are used here to denote the elements of \( \mathcal{N} \), and \( \bar{x} \) to represent a tuple of name variables.

Each channel has an arity, a fixed size for the tuples that may be passed along the channel. It is assumed that an implicit type system enforces the consistent use of arities, that each name is associated with a type, and that there are infinite many names for each type.

The following grammar defines the categories of processes, join-patterns and definitions:
\[ P \text{ def} = \begin{cases} \text{processes} & \text{null process} \\ 0 & x(\vec{v}) \text{ message} \\ \text{def } D \text{ in } P & \text{local definition} \\ P | P & \text{parallel composition} \end{cases} \]

\[ J \text{ def} = \begin{cases} \text{join-patterns} & x(\vec{v}) \text{ message pattern} \\ J | J & \text{join of patterns} \end{cases} \]

\[ D \text{ def} = \begin{cases} \text{definitions} & J \triangleright P \text{ reaction rule} \\ D \land D & \text{conjunction of definitions} \end{cases} \]

A process \( P \) is either an emission of an asynchronous polyadic message \( x(\vec{v}) \), a definition \( \text{def } D \text{ in } P \), or a parallel composition of processes \( P | P \). Intuitively, \( x(\vec{v}) \) is an asynchronous message that sends the tuple of values \( \vec{v} \) on \( x \). A local definition \( \text{def } D \text{ in } P \) defines the process \( P \) in the scope \( D \). The null process \( 0 \) does nothing.

A definition \( D \) consists of one of several elementary definitions \( J \triangleright P \), where the process \( P \) is guarded by the join-pattern \( J \).

Join-patterns entirely describe the operations of the defined names. A join-pattern is a list of message patterns of type \( x(y_1, \ldots, y_n) \), where the received variables, \( y_1, \ldots, y_n \), are all distinct and bound by the message pattern, and the channel \( x \) is defined and thus also bound by the join-pattern. Channels in the Join Calculus may be introduced only in join-patterns. Basically, \( J \triangleright P \) says that \( P \) may run whenever there are messages that match the join-pattern \( J \). In this case, the received variables in the join-patterns are instantiated by the corresponding channels in the messages. Definitions of form \( D \land D' \) are also allowed, meaning that whenever one of the join-patterns in the conjunction of definitions is matched, the corresponding process may run. The same channel may appear in several components of the disjunction. Definitions obey lexical scoping rules, and names defined in \( D \) are bound in the whole expression \( \text{def } D \text{ in } P \).

The defined port names in the join pattern \( J \) in the expression \( \text{def } J \triangleright P \text{ in } Q \) bind the names that appear in \( P \) and \( Q \). Received variables \( rv(J) \), defined variables \( dv(J) \) and \( dv(D) \), and free variables \( fv(D) \) and \( fv(P) \), are specified by structural induction as follows:

\[
\begin{align*}
rv(x(\vec{v})) & \text{ def} = \{ u \in \vec{v} \} \\
rv(J | J') & \text{ def} = rv(J) \cup rv(J') \\
dv(x(\vec{v})) & \text{ def} = \{ x \} \\
dv(J | J') & \text{ def} = dv(J) \cup dv(J') \\
dv(J \triangleright P) & \text{ def} = dv(J) \\
dv(D \land D') & \text{ def} = dv(D) \cup dv(D')
\end{align*}
\]
Join patterns are linear, and thus no received or defined variables may appear twice in the same pattern $J$. A name $x$ is fresh in $P$ if $x \notin fv(P)$.

### 2.5.2 Operational Semantics

The semantics of the Join Calculus operates on higher-order solutions, expressions of form $R \vdash M$ where $R$ is a multiset representing the current reduction rules, and $M$ is a multiset of molecules representing the processes running in parallel.

Rules are either of kind heating/cooling, written $\Rightarrow$, or reduction rules, written $\rightarrow$. Rules of type $\Rightarrow$ correspond to the underlying structural equivalence on processes, whereas the single reduction rule $\rightarrow$ expresses the mechanism of communication.

The rules are the following:

- **(str-join)**
  \[
  P \parallel Q \quad \vdash \quad \Rightarrow \quad P, Q
  \]

- **(str-and)**
  \[
  D \land E \quad \vdash \quad \Rightarrow \quad D, E
  \]

- **(str-def)**
  \[
  \vdash \quad \text{def} \ D \ \text{in} \ P \quad \Rightarrow \quad P_{\sigma_{dv}} \quad \vdash \quad P_{\sigma_{dv}}
  \]

- **(red)**
  \[
  J \triangleright P \quad \vdash \quad J_{\sigma_{rv}} \quad \rightarrow \quad J \triangleright P \quad \vdash \quad P_{\sigma_{rv}}
  \]

Substitutions $\sigma_{dv}$ instantiate the $dv(D)$ in $R \vdash M$ to distinct, fresh names, and substitutions $\sigma_{rv}$ substitute the transmitted values for the distinct received variables in $rv(J)$. In these substitutions we implicitly assume $\alpha$-renaming of non-free variables to avoid name clashes.

Rules str-join and str-and express that parallel composition resp. definition conjunction are commutative and associative. The rule str-def describes how molecules defining new names and reactions are “heated”. The substitution $\sigma_{dv}$ enforces the static scope of definitions and plays a role similar to the renaming of bound names in the scope extrusion of restricted names in the $\pi$-calculus. The last rule, red, expresses reduction, where a molecule $J_{\sigma_{rv}}$ is consumed and substituted by a fresh copy of the guarded process $P$ with the received variables instantiated by the transmitted names.

### 2.5.3 The Distributed Join Calculus

In the distributed version of the reflexive CHAM, DRCHAM, processes and definitions are partitioned into several local solutions, thus yielding a flat hierarchy of solutions. Both local computation and global communication are representable in the model. Furthermore, creation and migration of local solutions are made possible by adding more structure. Location names are attached to solutions and organised as a tree of nested locations.

The DRCHAM is a multiset of CHAMs. The global state consists of several solutions $R_i \vdash P_i$ separated by $||$. Local solutions evolve internally as before, but now they may also interact according to the new reduction rule.
Thus, a message emitted in a given solution on a port defined at another solution may be forwarded to the solution where it is defined. In a well-formed DRCHAM every name is defined in at most one solution.

The DRCHAM is then extended with a set $\mathcal{L}$ of location names, ranged by $a, b, \ldots \in \mathcal{L}$. The set $\mathcal{L}^*$ of location strings is ranged by $\phi, \psi, \ldots \in \mathcal{L}^*$. Running locations are local labeled solutions $R \vdash \phi P$. A location $\vdash \phi P$ is a sublocation of $\vdash \phi P$ if $\psi$ is a prefix of $\phi$. A DRCHAM is now defined as a multiset of distinct, prefix-closed labeled solutions that are uniquely identified by their rightmost location name, if any. Thus, solutions ordered by the sublocation relation form a tree. Location names obey the static scoping discipline, and can be created locally and passed in communications. The syntax of definitions is extended with a new location constructor

$$D \overset{\text{def}}{=} \ldots | a[D : P]$$

The meaning of this construct is the creation of a sublocation within the current location containing the unique definition $D$ and the unique running process $P$. This is expressed by the new structural rule

$$\text{str-loc } a[D : P] \vdash \phi \Rightarrow \vdash \phi \parallel \{D\} \vdash \phi \{P\} \ (\text{a frozen})$$

The side condition implies that there is no solution of the form $\vdash \phi \psi$, thus guaranteeing that the whole subtree of sublocations of $a$ is captured by the definition $D$. All reactions defining one name belong to a single location in a well-formed DRCHAM.

The DRCHAM is extended with a new primitive for migration, along with a new chemical solution

$$P \overset{\text{def}}{=} \ldots | \text{go}(b, k)$$

$$\text{move } a[D : P] \mid \text{go}(b, k) \vdash \phi \parallel \{a[D : P] \mid k()\} \vdash \phi \psi$$

This may be seen as a migration of location $a$ at $\phi a$ to a new position $\phi b a$, where the destination is identified by its relative name $b$. The continuation $k()$ can then trigger other computations.

### 2.5.4 Binding in the Join Calculus

The static scoping discipline of the Join Calculus definitions forbids overloading (see [47] for details). In the Join Calculus, the semantics of channels is governed by the reaction rule binding the name which must occur within its scope. Unbound names in the Join Calculus are meaningless, and all names must occur within the scope of a reaction pattern binding the name. In a distributed open environment this scheme would hinder communication among distinct parties. To remedy this, the scoping rules are softened in the distributed version of the calculus [48].

In the distributed version of the Join Calculus, communication between distinct parties is made possible by resorting to a centralised name server, and assuming that agents on different machines share some naming conventions. However, in the
absence of dynamic binding, the external name server must be sometimes invoked even when a process is running within the location where a name was declared, e.g. when the process has migrated from another location and wants to access local resources. Dynamic binding is nevertheless present, albeit only implicitly, in the shared naming conventions used to bootstrap the network, although with an extra and unnecessary level of indirection. Shared names are introduced indirectly as strings, cf. the way strings are use for dynamic dispatch in the functional language Scheme [6].

An example from the Join Calculus tutorial may illustrate this point. ¹

A function for squaring integers is defined in a location named \textit{here} with internal name \textit{f}, and then registered in the name server \textit{ns} where it is with the name “\textit{square}”, a string, as follows:

\begin{verbatim}
# let f(x) = # print_string(["\text{ml.string_of_int(x)}"])
# reply x*x # do ns.register("square",f)
\end{verbatim}

The location \textit{here} is also defined and registered in the name server as “\textit{here}”:

\begin{verbatim}
# loc here end # do ns.register("here",here)
\end{verbatim}

On another machine, a location \textit{mobile} wraps a loop computation involving calls to a function \textit{sqr}, an internal name for the function square, and \textit{sum}. In order to execute the loop, the process migrates to location \textit{here}, where \textit{square} is defined, by looking in the name server for the entries corresponding to the string “\textit{here}”:

\begin{verbatim}
# loc mobile # init # let here = ns.lookup("here") in # go(here); # let sqr = ns.lookup("square") in # let sum(s,n) = # reply (if n = 0 then s else sum(s+sqr(n),n-1)) in # let result = sum(0,5) in # print_string("q: sum(5)= "^\text{ml.string_of_int(s)}^\text{"n"});
# end
\end{verbatim}

The interesting point here is that the binding of \textit{square} must be looked up in the name server, although it has been declared by the location \textit{here}, within which the process asking for the binding is running. A local call dynamically binding the string \textit{square} would be enough. In fact, what we have here is a disguised form of dynamic. Both locations must share the name and semantics of \textit{square}, which is in effect a public name, before migration. If no such common knowledge is assumed, \textit{mobile} would not be able to use any resources provided by \textit{here}.

Dynamic dispatch is encoded in a similar way in many encodings of objects in the \(\pi\)-calculus and related formalisms [180, 135, 158, 130]. Basically, the name of a

¹Available at the URL http://join.inria.fr
channel, which in this case is not intended to be used as a channel, is matched, i.e. compared, to a name which has been imported via a message passing, and if the matching yields the value true then the corresponding method is invoked. This shows that matching in the $\pi$-calculus may play a role similar to dynamic binding. This may help explain many of the rather singular features associated with the matching operator in the $\pi$-calculus, which converts names into something more than abstractions for channels. In Chapter 3 we show also that matching may be expressed in terms of an operator that encode the notion of dynamic binding.

The computational model of the reflexive CHAM is thus insufficient for modelling communication in open systems because of its rigid static binding discipline. Nevertheless, it is a useful notation for modelling object-based systems if we introduce dynamic binding. In Section 5.2 we define such a notation.

### 2.6 LLinda - Locality Based Linda

LLinda [35] is an coordination language with a asynchronous communication mechanism based on a shared global environment. In Linda processes interact by asynchronously dropping and picking tokens from a common space consisting of a multiset of tuples, sequences of values of expression that may contain variables. Tuples are selected by pattern-matching. There are four primitives for manipulating tuples: $\text{out}(t)$ to add tuples to the tuple space; $\text{eval}(t)$ to both add a new tuple and create a new concurrent process to evaluate $t$; $\text{in}(t)$ for retrieving and removing $t$ from the tuple space; and $\text{read}(t)$ for reading $t$ without withdrawing it from the tuple space. The operation is suspended if no matching tuple is found.

LLinda [43] is a distributed version of Linda extended with explicit localities, a CCS-like process calculus, and support for process migration. Groups of processes and data located in a determined space or locality may be viewed as distinct entities. Encapsulation may be modelled as a tuple space located at a single locality. LLinda may be seen as an asynchronous value-passing process calculus. It is endowed with multiple distributed tuple spaces with a flat structure, i.e. localities may not be nested. LLinda is basically a higher-order calculus where processes may be passed between locations, and binding of free identifiers representing localities may be either static or dynamic. The basic actions of LLinda are similar to Linda, but enriched with explicit locality information. Data and processes may be retrieved from different nodes.

The syntax of LLinda assumes a set of localities or nodes, where processes and tuples are allocated, as well as a set of logical localities, symbolic names for sites including $\text{self}$, used by processes to denote the locality at which they are presently running. Nodes are endowed with a unique locality name and an assignment $\gamma$ from logical localities to localities. This assignment associates an environment with a locality. LLinda may thus be considered as a nominal calculus, with the names of the localities playing the same role as communication channels in the $\pi$-calculus. The most important difference is that not only names and localities, but processes as well are seen first-class data that can be manipulated, generated and passed between locations as any other data occurring in tuples. The process expression $P(\gamma)$ denotes a closure for $P$, i.e. the process $P$ packaged with the allocation of logical localities specified by $\gamma$. If $P(\gamma)$ is executed in a locality with an assignment $\gamma'$, a free location variable $u$ occurring in $P$ and belonging to the domain of $\gamma$ is bound to $\gamma(u)$ rather than to $\gamma'(u)$. If a closure is not part of a process expression, the binding of location variables is dynamic. A static binding discipline is adopted for $\text{out}$ operations, whereas for remote $\text{eval}$ operations the binding discipline is dynamic. Thus, if a process $P$ located at $\ell_1$ spawns a process $Q$ at the remote
As noted by the authors, “localities in LLinda can be used for simulating private name passing and the scope extrusion mechanisms of the π-calculus,” and consequently “a natural encoding of the asynchronous π-calculus in LLinda can be easily programmed.” The important question whether there is any natural encoding of LLinda in the π-calculus is nevertheless not considered. As we shall see in Chapter 3, in the presence of higher-order constructs it is hard to encode dynamic binding in the π-calculus.

### 2.7 The Seal Calculus

The Seal calculus \([175]\) is a distributed process calculus supporting the notion of location and movement of computational entities. It is described as “the π-calculus with hierarchical location mobility and resource access control.” Its goal is “to expose the network” by making localities visible and “to hand over control of localities and low-level protection to the system programmer.” This is accomplished by the introduction of mobility and protection primitives.

The Seal Calculus provides also a hierarchical protection model. Each level in the hierarchy can implement security policies by mediation. This means that a higher level may scrutinise and control observable actions performed at a lower level. Furthermore, the calculus has a notion of protection boundary, which implies that operations crossing a boundary are syntactically different from local operations.

The Seal Calculus supports the notion of locations, processes and resources. Locations represent physical places and embody logical boundaries such as protection domains, sandboxes and applications. The process abstraction refers to flow of control, and resources to services.

Names denote seals and channels, and may be exchanged in communication and created dynamically as in the π-calculus. Channels are the only resources in the calculus, and are used to synchronise concurrent processes. Channels are also located, as processes. Names may be viewed as capabilities, and processes may only use the names they know.

Processes include the null process, processes prefixed by actions, process composition, replication, guarded replication, and a seal, i.e. a named location containing a process.

Seals are named locations structured hierarchically. The expression \(n[P]\) denotes a process \(P\) running at location \(n\). Channel notations specify the location where they are located. Thus, in the process

\[
n_1[P_1 | n_2[P_2 | n_3[P_3]]]
\]

a channel \(x\) located in \(n_2\) is denoted \(x^{n_2}\) by \(P_1\), \(x^*\) by \(P_2\), and \(x\uparrow\) by \(P_3\).

Communication in the Seal Calculus may be local or remote. Local communication takes place within a single seal between two co-located process over a local channel. Remote interaction, on the other hand, may be of two types: up-sync, where a process located in a parent seal synchronises with a process in a child seal along a channel located in the latter; and down-sync, where synchronisation between a parent process and a child process takes place along a channel located in the parent seal.

A seal may also be the object of communication. If \(y\) is a seal name, the expression \(\pi(y)P\) denotes a process ready to send a child seal \(y\) along \(x\). This kind of seal
mobility is called *objective*, since a seal is always moved by the parent. Computation encapsulated within a seal’s boundary is not affected by mobility. A receive action may be expressed by a non-binding expression \( x^\eta (\vec{y}) P \), where \( \eta \) is either \( s \), \( s \) (a seal name) or \( \uparrow \), and \( \vec{y} = y_1 \ldots y_n \). This process is ready to receive one seal and create \( n \) identical copies of it. For instance, the process

\[
\Phi'(n).0 \mid y^s(n_1,n_2). (Q \mid n[P])
\]

may reduce to

\[
Q \mid n_1[P] \mid n_2[P]
\]

A protection mechanism called *portal* is included in order to protect inter-seal communication. The idea is that a channel located in a seal may only be used previous opening of a portal via the action open, \( x \), executed within the seal, where \( x \) denotes either a positive or negative occurrence of the channel \( x \).

Finally, total *mediation* is implemented by a security policy that controls all observable actions involving any child, and by inspection of all values exchanged in a communication. For details about the syntax and semantics of the Seal Calculus we refer to Vitek [175].

The seal model does not rely on a global state. Access to local resources may be restricted. Localities are explicit and communication is restricted to local or parent-child interaction. Dynamic reconfiguration is supported by mobility, name passing and dynamic binding. Mobility supports seal migration and seal duplication, and implicit configuration is obtained by the dynamic binding of the \( \uparrow \) operator, which denotes the parent environment.

The seal model may be described as a higher-order calculus with dynamic binding. Accordingly, translating the Seal Calculus to a first-order calculus may be hard. As in LLinda (see Section 2.6), the issue has not been given a more extensive treatment. It is pointed out that Sangiorgi’s technique [149] for translating a higher-order \( \pi \)-calculus into a first-order one fails because seal mobility is defined on running processes, whereas in Sangiorgi’s translation the process intended for transmission is prefixed by the trigger and replicated. However, the fact that the dynamic binding of the \( \uparrow \) operator would also cause Sangiorgi’s translation to fail is not mentioned.

### 2.8 Other Related Calculi and Systems

The Spi Calculus is an extension of the \( \pi \)-calculus with cryptographic primitives [4]. In analogy to this calculus and the \( \pi \)-calculus, the Sjoin Calculus is an extension of the Join Calculus with basically cryptographic primitives [3]. The Blue Calculus is a direct extension of both the \( \lambda \)-calculus and the \( \pi \)-calculus [26]. It is a direct model for higher-order, untyped concurrency with the same expressive power as the \( \pi \)-calculus. Its semantics is operational and follows the CHAM style.

Concerning systems, particularly object-oriented ones, the following are especially relevant for this work.

Obliq [29] is a lexically-scoped, untyped and interpreted language supporting distributed object-oriented computation. Distributed lexical scoping is the key mechanism for managing distributed computations, and the binding location of every identifier is determined by the text surrounding the identifier. Free identifier of network-transmitted procedures are thus bound to their original locations, as expected in an static binding discipline.
Telescript [184] is an object-oriented class-based language with runtime typing designed for network programming. In contrast to Mobile Ambients, in Telescript agents may move but places are static. Nesting of places is allowed. Users can create their own places nested within other existing places.

Java [19] is a class-based object-oriented language with an emphasis on portability and security. The language provides a working paradigm for mobile computation and is very widespread today. It includes the “Applet” model. Applets are small applications that can be automatically downloaded and executed.

Objective Caml (O’Caml) [95] is a functional language in the ML tradition originating from Caml[139]. It includes support for concurrency through threads and mutexes, imperative features, and a class-based object system integrated within a functional core.

Objective Linda [85] is a coordination model combining object-oriented with uncoupled generative communication, and designed for the needs of object-oriented parallel programming. The language provides hierarchies of object spaces that may be accessed by a generative matching mechanism. Linda’s notion of tuples is substituted by objects, which are instances of abstract data types, and tuple spaces are substituted by object spaces. Active agents are able to access object spaces in which objects are stored as encapsulated data items. Object spaces are the only kind of entities shared between objects, and objects are either moved or copied whenever transferred between agents and object spaces. There is no sharing between objects.
Chapter 3

Dynamic Binding in the \(\pi\)-calculus

One of the most significant contributions of the \(\pi\)-calculus has been the demonstration that higher-order features in concurrency can be eliminated in favour of first-order ones by means of channel name generation and communication. It has been argued that higher-order features are matters of convenience only and that no essential descriptive or analytical power is added by the higher-order features.

In this chapter we reexamine this position. We show that some applications of higher-order processes require better control of communication capabilities than what is provided by the \(\pi\)-calculus primitives. We investigate the consequences of adding dynamic restriction to the first- and higher-order \(\pi\)-calculus, and show that in the presence of dynamic binding the current methods for reducing higher-order communication features to first-order ones breaks down. We show, as our main result, that the higher-order reduction can be regained, using an approach by which higher-order communications are replaced by the transmission and dynamic interpretation of syntax trees. However, the reduction is very indirect, and not usable in practice. This throws new light on the position that higher-order features in the \(\pi\)-calculus are superfluous and not needed in practice.

3.1 Introduction

The issue concerning the reducibility of higher-order \(\pi\)-calculus to first-order has been extensively studied in the context of \(\lambda\)-calculus under various evaluation regimes (cf. [110]), and in his thesis [149] Sangiorgi explored in depth the reduction of higher-order processes to first-order ones. Instead of communicating a higher-order object, a local copy is created, protected by a trigger in the shape of a newly generated channel name. This trigger can then be communicated in place of the higher-order object itself. On the basis of this sort of reduction it has been argued (cf. [149]) that, in the context of the \(\pi\)-calculus, higher-order features are matters of convenience only: No essential descriptive or analytical power is added by the higher-order features.

Here we reexamine this position, and find it holds in principle, but not in practice. Practical applications call for process combinators other than those provided by the basic \(\pi\)-calculus. Specifically we consider the dynamic restriction, or blocking operator of Thomsen’s CHOCS [168] (see Section 1.3.4 for the motivation). Adding blocking to the higher-order \(\pi\)-calculus causes Sangiorgi’s reduction to break down. In the presence of blocking it remains possible to reduce the higher-order calculus
to the first-order one in a compositional manner. The reduction, however, is complicated, and amounts in effect to the communication and interpretation of parse trees. In contrast to Sangiorgi's reduction which is conceptually quite simple this reduction can not be used in practice to reduce non-trivial arguments concerning higher-order processes to arguments concerning first-order ones. The reduction is very general and can be applied to a wide range of static process combinators.

Our interest in the blocking operator stems from some difficulties connected with the representation of cryptographic protocols in the higher-order π-calculus [39].

3.1.1 Cryptographic Protocols

The purpose of cryptographic protocols is to ensure secure communication over insecure channels using cryptographic techniques. These protocols operate in an open-system environment consisting of an arbitrary number of hosts or parties referred to as principals in the security literature, and their goal is largely to provide authentication and/or secrecy. An association network model with end-to-end data paths or connections through the network is usually adopted. In this model intruders are represented as hostile principals that may position themselves in any point of the network, initiate new connections, intercept any messages, make replays of old messages, etc.

One of the main assumptions in this model is that the ends of association are secure areas which are not subject to attack, and that encryption and decryption take place in this safe area. Principals should thus possess a private region for safe encryption and decryption.

To illustrate it we may look at the modified version of Andrew's authentication protocol presented in [28], whose purpose is the establishment of mutual authentication between an initiator A and a responder B, as well as the negotiation of a new fresh key for a secure communication session.

In this protocol a fresh key $K_{ab}'$ is exchanged for a communication session between A and B provided that these principals initially share a secure private key $K_{ab}$. Principal A, the initiator of the communication, announces its desire to start a new session with B by sending B a so-called nonce, $N_a$, defined as a message assumed to be fresh and unpredictable, usually a random number, encrypted by $K_{ab}$. We let $\{m\}_K$ denote a message $m$ encrypted with key $K$, which may also be a compound of $n$ messages separated by commas: $m = m_1, \ldots, m_n$. Thus $\{N_a\}_K$ denotes the message consisting of the nonce $N_a$ encrypted with the key $K_{ab}$. Message 1 below describes this step, indicating also that A is the sender of the message $\{N_a\}_K$, and thus the initiator of the protocol who seeks to establish a session, and that the message is aimed at B, the responder.

If B receives this message, it will be able to decrypt it in a secure way since the protocol assumes that B is in possession of the key $K_{ab}$. Upon reception of A's request, B returns, encrypted by $K_{ab}$, the nonce $N_a$, in order to check the freshness of the message, along with a new session key $K_{ab}'$. After receiving this message A should be assured that it is talking to B since A assumes that the private key $K_{ab}$ is a shared secret between itself and B, and thus only B would have been able to decrypt $\{N_a\}_K$, and in this way get hold of the clear text of the nonce $N_a$. After decryption of $\{N_a, K_{ab}'\}_K$, A returns the nonce $N_a$ encrypted by the new key generated by B, $K_{ab}'$ (message 3). After receiving this message B should be assured that it is talking to A, since B assumes that only A would be able to decrypt $\{N_a, K_{ab}'\}_K$ and in this way to obtain $K_{ab}'$. 

27
The protocol can be described in this standard notation as follows:

1. $A \rightarrow B : \{N_a\}_{K_{ab}}$
2. $B \rightarrow A : \{N_a, K_{ab}'\}_{K_{ab}}$
3. $A \rightarrow B : \{N_a\}_{K_{ab}'}$

3.1.2 Modeling Cryptographic Protocols in the Higher-Order $\pi$-calculus

With some simplifications, the higher-order $\pi$-calculus can be used to model cryptographic protocols in the following way. Keys are treated as names, and encrypted messages as a special kind of process. Thus, a message $m$ encrypted with key $K$ may be represented in the higher-order $\pi$-calculus as a process:

$$\overline{K}(m).0$$

In this process, $m$ may be passed along output port $K$, which represents the key, and may be “read” or received by any process in possession of the key $K$. Thus, the Andrew protocol described above may be specified in the higher-order $\pi$-calculus as follows:

1. $A \rightarrow B : \overline{K_{ab}}(N_a).0$
2. $B \rightarrow A : \overline{K_{ab}}(N_a, K_{ab}').0$
3. $A \rightarrow B : \overline{K_{ab}'}(N_a).0$

In message 1, $A$ sends $B$ a second-order process $\overline{K_{ab}}(N_a).0$, representing message $N_a$, the nonce, encrypted by $K_{ab}$. Provided that $B$ possesses $K_{ab}$, it will be able to get hold of the nonce and transmit it to $A$ encrypted by $K_{ab}$, together with a new session key $K_{ab}'$. This action corresponds to the transmission of the process $\overline{K_{ab}}(N_a, K_{ab}').0$. Finally, after decryption of the message received from $B$, $A$ transmits the process $\overline{K_{ab}'}(N_a).0$ representing encryption of nonce $N_a$ by $K_{ab}'$.

Assuming that $A$ and $B$ share an insecure channel $\text{xf}e r$, the behaviour of both principals may be specified in the higher-order $\pi$-calculus as follows:

$$A \overset{\text{def}}{=} (\nu N_a)(\text{xf}e r(\overline{K_{ab}}(N_a).0).\text{xf}e r(X). (X \mid K_{ab}(N, K_{ab}')(N = N_a.\overline{\text{xf}e r(\overline{K_{ab}'}(N_a).0).A'}))$$

$$B \overset{\text{def}}{=} \text{xf}e r(X). (X \mid K_{ab}(N). (\nu K_{ab}').\overline{\text{xf}e r(\overline{K_{ab}}(N, K_{ab}').\text{xf}e r(X). (X \mid K_{ab}''(N'').[N'' = N].B']))).$$

Here $A'$ and $B'$ represent the behaviour of $A$ resp $B$ after the exchange of the session key and are left unspecified.

The complete specification of the protocol is

$$\text{SPEC} \overset{\text{def}}{=} (\nu K_{ab})(A \mid B).$$

As far as encrypted message passing is concerned, the specification of the protocol in the $\pi$-calculus presents no difficulties, but the act of decrypting those messages, represented explicitly in the specification by a channel that represents the encryption key, is only implicit in the original protocol, in which it is assumed that each principal may decrypt securely any message whose encryption key it knows. This requirement is not met by the specification. Assume that the key $K_{ab}'$ becomes compromised because of a flaw in the protocol. This could happen by assuming the protocol is enacted in a possibly hostile environment $Z$:

$$\text{SPEC} \mid Z$$

Upon reception of $\{N_a\}_{K_{ab}'}$ (message 3), $B$ proceeds in the following way:
\[ \overline{K_{ab}^\dagger(N_a)0} | K_{ab}^\dagger(N').[N' = N_a]B'. \]

Now, due to lack of local control over communication channel \( K_{ab}^\dagger \), if \( Z \) knows \( K_{ab}^\dagger \), it could snatch nonce \( N_a \) from \( B \) during decryption, contrary to one of the basic assumptions of cryptographic protocols. In this particular case nothing would be gained by the intruder, since if he already is in possession of the key he could have intercepted and decrypted it himself. But there are protocols where a message \( m \) might be doubly encrypted, a feature that is common in public-key based protocols. For instance, a message \( m \) may be encrypted first by \( A \)'s public key \( K_a \) for authenticity, becoming \( \{m\}_K \), and then by \( B \)'s secret key \( K_b \) for privacy, becoming \( \{\{m\}_K\}_K \). This message would be encoded in the \( \pi \)-calculus by the process
\[ \overline{K_b(K_a(m)0)0} \]

The behaviour of \( B \) under reception of the message could be specified as follows:
\[ xfer(X).((X | (K_b(Y)(Y | K_a(M).A'))) \]

Now, any intruder \( Z \), though not knowing \( B \)'s secret key \( K_b \), would be able to steal the message \( m \) from \( B \), once it has reached the configuration \( \overline{K_a(m)0} | K_a(M).A' \), since key \( K_a \) is public and thus may be assumed to be known by \( Z \). Once again, decryption is shown to be unsafe, contrary to assumption.

### 3.1.3 Extension of the \( \pi \)-calculus with Blocking

The shortcoming mentioned in the previous section suggested the extension of the \( \pi \)-calculus with an operator that we call blocking and is denoted by \( P \setminus a \). It works as a kind of firewall preventing communication along the channel \( a \) between \( P \) and its environment. This operator allows a transition \( P \setminus a \xrightarrow{\alpha} P' \setminus a \) only if \( P \xrightarrow{\alpha} P' \) and \( a \) is not the subject of the action \( \alpha \). It is nevertheless possible that \( a \) is the object of an output action \( \alpha \), thus allowing communication of \( a \) across the firewall. In contrast to the restriction operator, which restricts the scope of of both action subjects and action objects, blocking merely restricts the communication capabilities of channels.

The behaviour of \( B \) during decryption of \( m \) can be specified as
\[ (\overline{K_a(m)0} | K_a(M).A') \setminus K_a, \]
thus protecting \( m \) from being snatched by \( Z \).

Blocking is basically the dynamic restriction operator of Thomsen's CHOCS [168].

Another attempt to describe and analyse security protocols with the help of the \( \pi \)-calculus is the spi calculus ([5, 4]). The authors also found it necessary to extend the \( \pi \)-calculus by introducing cryptographic primitives. In contrast, our approach yields a system that is more general and may also have application in other fields. Therefore, this study is dedicated to analysing the impact of extending the standard \( \pi \)-calculus with blocking and is independent of the application that inspired it.

The main results of this study is that blocking enhances the expressiveness of the standard \( \pi \)-calculus and is equivalent to this calculus extended with mismatch, and that in the presence of blocking it is still in principle possible to reduce the higher-order \( \pi \)-calculus to the first-order \( \pi \)-calculus, although the reduction is sound but not complete, and thus not fully abstract.

### 3.1.4 Overview of the Chapter

In Section 3.2 we will focus on the extension of the first-order \( \pi \)-calculus with blocking. Then, in Section 3.3, we study that extension in the context of a higher-
order setting. A reduction from the higher-order to first-order \( \pi \)-calculus with blocking is also shown. Finally, in the last section we present some conclusions.

3.2 The First-Order \( \pi \)-calculus with Blocking

We study the extension of the first-order \( \pi \)-calculus, as described in [112], with blocking, similar to the dynamic restriction operator of CHOCs [168], and akin to the CCS restriction operator.

The blocking operator is studied in the context of the first-order \( \pi \)-calculus with late instantiation as in [112]. Many laws, in a slightly modified form, associated with the restriction operator in CCS are shown to be valid for the blocking operator. The algebraic theory in [112] is enlarged with new axioms that preserve soundness and completeness for finite agents, and it is shown that the blocking operator and mismatch may be expressed in terms of each other.

Below, \( \pi \) denotes the language of the first-order \( \pi \)-calculus, \( \pi B \) the language obtained from \( \pi \) by adding blocking, and \( \pi M \) the language obtained from \( \pi \) by adding mismatch.

3.2.1 Agents and Transitional Semantics

Agents

We adopt the same name discipline as in [112]. Accordingly, we assume an infinite set of names \( \mathcal{N} \), use \( x, y, z, u, v, w, u \) as metavariables over names, and assume a set of agent identifiers where each agent identifier \( A \) has a nonnegative arity.

**Definition 2.1 (Agents)** The agents of \( \pi B \) are generated according to the following abstract syntax, using \( P, Q, R \) as metavariables over agents:

\[
P ::= 0 \mid \pi y.P \mid x(y).P \mid \tau.P \mid (x)P \mid [x = y]P \mid P \mid P + Q \mid A(y_1, \ldots, y_n) \mid P\backslash z
\]

Here 0 is a nullary operator and \( n \) is the arity of \( A \).

All operators, except blocking, are familiar from the \( \pi \)-calculus. For blocking we use the CCS restriction operator notation: \( P\backslash z \) blocks communication between \( P \) and its environment along the channel \( z \) while allowing communication along other channels to mention \( z \).

Most definitions are the same as in [112]. The scope of the occurrence of \( z \) in \( P\backslash z \) is \( P \). An occurrence of \( z \) in \( P\backslash z \) is free, unless it occurs within the scope of prefix \( x(z) \) or the restriction operator \( (z) \). The set of names occurring free in \( P \) is denoted \( \text{fn}(P) \), and \( \text{fn}(P) \cup \text{fn}(Q) \cup \cdots \cup \{x, y, \cdots\} \) is sometimes used as shorthand for \( \text{fn}(P) \cup \text{fn}(Q) \cup \cdots \cup \{x, y, \cdots\} \). The set of bound names of \( P \) is denoted \( \text{bn}(P) \) and the set of names in \( P \), written \( n(P) \), is defined as \( \text{fn}(P) \cup \text{bn}(P) \).

We assume the existence of a unique defining equation for each agent identifiers \( A \) of arity \( n \) of the form

\[
A(x_1, \cdots, x_n) \overset{\text{def}}{=} P,
\]

where the \( x_i \) are pairwise distinct, \( \text{fn}(A(\overline{x})) \subseteq \{x_1, \cdots, x_n\} \) and \( \text{bn}(A(\overline{x})) = \text{bn}(P) \), assumed to be finite, where \( \overline{x} = x_1, \cdots, x_n \).

A substitution is defined as a function \( \sigma \) from \( \mathcal{N} \) to \( \mathcal{N} \) precisely as in [112], with the following addition: \( (P\backslash z)\sigma \overset{\text{def}}{=} P\sigma\backslash z\sigma \).

The symbol \( \equiv_\alpha \) denotes the relation of alpha-convertibility on agents (note that the subscript \( \alpha \) bears no relation to actions \( \alpha \)).
Transitions

Transitions have the general shape $P \xrightarrow{\alpha} Q$, where the action $\alpha$ is either an input action of form $x(y)$, a free output action of form $\mathfrak{y}_y$, a bound output of form $\mathfrak{y}(y)$, or the silent action $\tau$. Free output actions and the silent action are called free actions, and input and bound output actions are called bound actions. The subject of an action $\alpha$, denoted $s(\alpha)$ is the singleton set containing $x$, and the object of $\alpha$ is either the free object $y$ in case $\alpha$ is of form $\mathfrak{y}_y$, or the bound object $y$ if $\alpha$ is of form $x(y)$ or $\mathfrak{y}(y)$. Actions of form $\tau$ and $\mathfrak{y}_y$ are called free actions. If $\alpha = \tau$, $s(\alpha)$ is empty. The set of bound names $bn(\alpha)$ of an action $\alpha$ is empty if $\alpha$ is a free action, otherwise it is the singleton set containing $y$ if $\alpha$ is $x(y)$ or $\mathfrak{y}(y)$. The set of free names $fn(\alpha)$ contains the elements of $s(\alpha)$ and the free object, if any, of $\alpha$. The names of $\alpha$, denoted $n(\alpha)$, is defined as $bn(\alpha) \cup fn(\alpha)$. We also use $\mathfrak{y}(y)$ as a shorthand for $(y)\mathfrak{y}$. The operational semantics is given in the TABLE 1. The only difference with the operational semantics of $\pi$ is the occurrence of the rule $\text{BLOCK}$.

![TABLE 1](image)

**Rules of Action**

<table>
<thead>
<tr>
<th>TAU-ACT:</th>
<th>OUTPUT-ACT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau. P \xrightarrow{z} P$</td>
<td>$\mathfrak{y}_y. P \xrightarrow{\mathfrak{y}} P'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT-ACT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(z). P \xrightarrow{n(w)} P[w/z]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUM:</th>
<th>MATCH:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{\alpha} P'$</td>
<td>$P \xrightarrow{[x = x]} P \xrightarrow{P'}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P{y/x} \xrightarrow{\alpha} P'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAR:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{\alpha} P'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COM:</th>
<th>CLOSE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{\mathfrak{y}} P'$</td>
<td>$P \xrightarrow{z(w)} P \xrightarrow{z(w)} Q'$</td>
</tr>
<tr>
<td>$P \xrightarrow{[y]P} P'[y/z]$</td>
<td>$P \xrightarrow{[y]} P \xrightarrow{P'[y]} Q'[y/z]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RES:</th>
<th>OPEN:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{\alpha} P'$</td>
<td>$P \xrightarrow{\mathfrak{y}} P'$</td>
</tr>
<tr>
<td>$(y) \xrightarrow{P} (y)P'$</td>
<td>$(y) \xrightarrow{P} P[w/y]$</td>
</tr>
<tr>
<td>$y \not\in n(\alpha)$</td>
<td>$w \not\in fn([y]P')$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLOCK:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{\alpha} P'$</td>
</tr>
</tbody>
</table>

*Obs: Symmetric forms for operators $+$ and $|$ have been omitted*
3.2.2 Strong Bisimilarity

The notion of late equivalence for $\pi$ presented in [112] is adopted here for $\pi B$.

Let the relation $\Rightarrow$ be the reflexive and transitive closure of $\xrightarrow{\tau}$, $\Rightarrow^{\alpha}$ be $\Rightarrow \cup \alpha \Rightarrow$, and $\Rightarrow^{\alpha}_{\text{exiv}} \Rightarrow$ be the same as $\Rightarrow^{\alpha}$ if $\alpha \neq \tau$, otherwise $\Rightarrow$. We have the following definition:

**Definition 2.2: Strong and Weak Bisimulation.** A binary relation $S$ on first-order agents is a (strong) simulation if $P \xrightarrow{\alpha} Q$ implies:

1. If $P \xrightarrow{\alpha} P'$ and $\alpha$ is a free action, then for some $Q'$, $Q \xrightarrow{\alpha} Q'$ and $P' SQ'$.
2. If $P \xrightarrow{\pi(y)} P'$ and $y \not\in n(P, Q)$, then for some $Q'$, $Q \xrightarrow{\pi(y)} Q'$ and for all $w$, $P' \{w/y\} SQ' \{w/y\}$.
3. If $P \xrightarrow{\pi(y)} P'$ and $y \not\in n(P, Q)$, then for some $Q'$, $Q \xrightarrow{\pi(y)} Q'$ and $P' SQ'$.

A binary relation $S$ on agents is a (strong) bisimulation if both $S$ and its inverse are simulations.

The relation $\cong$, (strong) bisimilarity, on agents is defined by $P \cong Q$ if and only if there exists a bisimulation $S$ such that $PSQ$.

A binary relation $R$ on first-order agents is a weak simulation if it is symmetric and satisfies the following property: $PRQ$ and $P \xrightarrow{\alpha} P'$, where $\text{bn}(\alpha)$ is fresh, implies that:

(i) If $\alpha = a(x)$ then $\exists Q'' : \ x : Q \Rightarrow^{a(x)} Q'' \land \forall u \exists Q' : Q'' \{u/x\} \Rightarrow Q' \land P' \{u/x\} \mathcal{R} Q'$

(ii) If $\alpha$ is not an input then $\exists Q' : \ x : Q \Rightarrow^{\alpha} Q' \land P' \mathcal{R} Q'$

The relation on agents $\cong$, weak bisimilarity, is defined by $P \cong Q$ if and only if there exists a weak bisimulation $\mathcal{R}$ such that $PRQ$.

3.2.3 Properties of Strong Bisimilarity

Here we study the properties of strong bisimilarity in $\pi B$. It turns out that the corresponding lemmas and theorems for $\pi$ in [112] are valid for $\pi B$. The proofs of most lemmas and theorems are the same, except for the cases involving the blocking operator, so we limit the proof to those and refer to [112] for the other cases.

**Transitions and Alpha-Conversion**

**Theorem 2.1** $\equiv_{\alpha}$ is a strong bisimulation.

**Proof:** see appendix.

**Bisimilarity as an Equivalence**

The results concerning bisimulation as an equivalence in [112] are valid for $\pi B$.

**Theorem 2.2:**

(a) $\cong$ is an equivalence relation.
(b) If $P \simeq Q$ then
\[
\begin{align*}
\alpha P & \simeq \alpha Q \quad (\alpha \text{ a free action}) \\
P + R & \simeq Q + R \\
[x = y]P & \simeq [x = y]Q \\
P \mid R & \simeq Q \mid R \\
(w)P & \simeq (w)Q \\
P \langle z & \simeq Q \langle z
\end{align*}
\]

Proof: See appendix.

Algebraic Laws for Bisimilarity

Most laws for the restriction operator have a corresponding law for the blocking operator, with slight modifications, while others involve both the restriction and the blocking operator.

Theorem 2.3:
The following equivalences hold:
\[
\begin{align*}
(a) & \quad (\alpha P) \langle z \simeq \alpha (P \langle z) & \text{ if } z \not\in n(\alpha) \\
(b) & \quad (\alpha P) \langle z \simeq 0 & \text{ if } s(\alpha) = z \\
(c) & \quad (P + Q) \langle z \simeq P \langle z + Q \langle z \\
(d) & \quad P \langle z \langle z' \simeq P \langle z z \\
(e) & \quad ((y)P) \langle z \simeq (y)(P \langle z) & \text{ if } y \neq z \\
(f) & \quad ([x = y]P) \langle z \simeq [x = y](P \langle z)
\end{align*}
\]

Proof: The following relations may be easily seen to be strong bisimulations:
\[
\begin{align*}
S_a & = \{(\alpha (P) \langle z, \alpha (P \langle z) \} \cup Id \\
S_b & = \{((\alpha P) \langle z, 0)) \}
\end{align*}
\]
\[
\begin{align*}
S_c & = \{((P + Q) \langle z, P \langle z + Q \langle z) \} \cup P, Q \text{ agents} \} \cup Id \\
S_d & = \{(P \langle z \langle z', P \langle z' \langle z) \} \} P \text{ agent } \\
S_e & = \{((y)P) \langle z, (y)(P \langle z) \} \} P \text{ agent } \\
S_f & = \{([x = y]P) \langle z, [x = y](P \langle z) \} \} P \text{ agent } \cup Id
\end{align*}
\]

3.2.4 Algebraic Theory

In this section we establish an axiomatisation of strong equivalence for $\pi B$ similar to the axiomatisation of strong ground equivalence for $\pi$ in [112].

Definition 2.3. The theory SGE consists of the following axioms and inference rules:

\begin{itemize}
  \item Alpha-conversion
  \item From $P \equiv Q$ infer $P = Q$.
  \item Congruence
  \item From $P = Q$ infer $\tau P = \tau Q$
\end{itemize}
\[ \forall y. P = \forall y. Q \]
\[ P + R = Q + R \]
\[ P \mid R = Q \mid R \]
\[ (x)P = (x)Q \]
\[ [x = y]P = [x = y]Q \]

C1 From \( P\{z/y\} = Q\{z/y\} \), for all names \( z \in \text{fn}(P,Q,y) \), infer
\[ x(y).P = x(y).Q \]

**Summation**

\[
\begin{align*}
S0 & \quad P + 0 = P \\
S1 & \quad P + P = P \\
S2 & \quad P + Q = Q + P \\
S3 & \quad P + (Q + R) = (P + Q) + R
\end{align*}
\]

**Restriction**

\[
\begin{align*}
R0 & \quad (x)P = P \quad \text{if } x \notin \text{fn}(P) \\
R1 & \quad (x)(y)P = (y)(x)P \\
R2 & \quad (x)(P + Q) = (x)P + (x)Q \\
R3 & \quad (x)(\alpha.P) = \alpha.(x)P \quad \text{if } x \notin n(\alpha) \\
R4 & \quad (x)\alpha.P = 0 \quad \text{if } x \text{ is the subject of } \alpha
\end{align*}
\]

**Match**

\[
\begin{align*}
M0 & \quad [x = y]P = 0 \quad \text{if } x \neq y \\
MR1 & \quad [x = x]P = P
\end{align*}
\]

**Expansion**

E Assume \( P \equiv \sum_i \alpha_i.P_i \) and \( Q \equiv \sum_j \beta_j.Q_j \), where no \( \alpha_i \) (resp. \( \beta_i \)) binds a name free in \( Q \) (resp. \( P \)): then infer
\[ P \mid Q = \sum_i \alpha_i.(P_i \mid Q) + \sum_j \beta_j.(P \mid Q_j) + \sum_{\alpha_i \text{ comp } \beta_j} \tau.R_{ij} \]
where the relation \( \alpha_i \) comp \( \beta_j \) (\( \alpha_i \) complements \( \beta_j \)) holds in four cases:

1. \( \alpha_i \) is \( \exists u \) and \( \beta_j \) is \( x(v) \); then \( R_{ij} \) is \( P_i \mid Q_j\{u/v\} \).
2. \( \alpha_i \) is \( \exists u \) and \( \beta_j \) is \( x(v) \); then \( R_{ij} \) is \( (w)(P_i\{w/u\} \mid Q_j\{w/v\}) \), where \( w \) is not free in \( (u)P_i \) or in \( (v)Q_j \).
3. \( \alpha_i \) is \( x(v) \) and \( \beta_j \) is \( \exists u \); then \( R_{ij} \) is \( P_i\{u/v\} \mid Q_j \).
4. \( \alpha_i \) is \( x(v) \) and \( \beta_j \) is \( \exists u \); then \( R_{ij} \) is \( (w)(P_i\{w/v\} \mid Q_j\{w/u\}) \), where \( w \) is not free in \( (v)P_i \) or in \( (u)Q_j \).

**Identifier**

I From \( A(\bar{x}) \overset{\text{def}}{=} P \) infer \( A(\bar{y}) = P\{\bar{y}/\bar{x}\} \).

**Blocking**

\[
\begin{align*}
H0 & \quad P\setminus y \mid z = P\setminus z \setminus y \\
H1 & \quad (P + Q)\setminus z = P\setminus z + Q\setminus z \\
H2 & \quad (\alpha.P)\setminus z = \alpha.(P\setminus z) \quad \text{if } z \notin s(\alpha) \cup \text{bn}(\alpha) \\
H3 & \quad (\alpha.P)\setminus z = 0 \quad \text{if } z \in s(\alpha) \\
H4 & \quad 0\setminus z = 0
\end{align*}
\]

34
**Blocking/Restriction**

\[
HR \quad ((x)P)\beta = (x)(P\beta) \quad \text{if } x \neq \beta
\]

**Blocking/Matching**

\[
HM \quad ([x = y]P)\beta = [x = y](P\beta)
\]

**Theorem 2.4 (Soundness):** If \( SGE \vdash P = Q \) then \( P \simeq Q \).

**Proof:** Soundness established in section 2.3 and in [112] Section 3.

**Theorem 2.5 (Completeness for finite agents):** For all finite agents \( P \) and \( Q \),

if \( P \simeq Q \) then \( SGE \vdash P = Q \)

**Proof:** see Appendix.

### 3.2.5 Expressiveness of the Blocking Operator

We now proceed to show that blocking and mismatch have the same expressive power in the language \( \pi BM \) (the first-order \( \pi \)-calculus with blocking and mismatch) with replication instead of agent constants. Thus, in the syntax definition of \( \pi BM \) we substitute \( P := !P \) for \( P := A(y_1, \ldots, y_n) \), and in the operational semantics definition of \( \pi B \) we substitute the transition rule \( IDE \) with the transition rule \( REP \):

**REP:**

\[
\begin{array}{c}
P \quad \frac{\alpha}{!P} \rightarrow P',
\end{array}
\]

The transition rule for mismatch, \( MISMATCH \), is as follows:

**MISMATCH:**

\[
\begin{array}{c}
P \alpha \rightarrow P',
\end{array}
\]

\[
[x \neq y]P \alpha \rightarrow P' (x \neq y)
\]

First we show that matching and mismatch may be expressed up to weak equivalence using blocking. Consider the agent \([x \neq y]P \in \pi BM\). This agent is equivalent to \( P \) if \( x \neq y \), and otherwise it is equivalent to \( 0 \). We let the agent \( P \) be guarded by a restricted channel \( w, w.P \), and be executed only if \( x \neq y \) by letting \( \bar{x}0 \) under blocking by \( y \) synchronise with \( x.\bar{w}0 \). This synchronisation takes place only if \( x \neq y \). To avoid additional communication capabilities, we block \( x \) too. Thus we obtain

**Proposition 2.1:**

\[
\exists y'[x \neq y]P \cong ([x \neq y]P \alpha \rightarrow ([x \neq y]P \alpha \rightarrow P).)
\]

**Proof:** If \( x = y \), then both \([x \neq y]P \) and \( ([x \neq y]P \alpha \rightarrow P) \) are equivalent to the null agent and are thus strongly equivalent. If \( x \neq y \), then the agent \( ([x \neq y]P \alpha \rightarrow P) \) will deterministically evolve into \( ([x \neq y]P \alpha \rightarrow P) \) after two silent transitions, and the latter is strongly equivalent to \( P \).

Matching may be expressed similarly:

**Proposition 2.2:**

\[
\exists y'[x = y]P \cong ([x \neq y]P \alpha \rightarrow P)
\]

The rest of this section is devoted to showing that blocking may be expressed up to strong equivalence by matching and mismatch.

By application of a simple transformation on agents \( T \), defined below, any agent containing occurrences of the blocking operator may be expressed up to strong equivalence by an agent without occurrences of blocking. The basic idea is to
eliminate occurrences of a blocked channel by replacing it by a fresh channel under the restriction operator. Since channels may be bound by an input prefix, we have to test them for equality with the blocked channel.

Let $P$ be any process. We define a transformation $T$ on agents that is a homomorphism over all operators except blocking, in which case it is defined as follows:

$$T(P \backslash z) = (w)T_{wz}(P), w \notin \text{fn}(P \backslash z)$$

For blocking $T$ is defined in terms of a transformation $T_{wz}$. Intuitively $T_{wz}(P)$ performs the task of testing the subject of an action prefix for equality with $z$ and in such case replace it by $w$. Assuming processes are conveniently renamed to avoid clash between $w, z$ and any other bound channel, $T_{wz}$ is defined as a homomorphism over all operators except blocking, input and output prefix:

$$T_{wz}(\tilde{x}y.P) = [x = z]\tilde{x}y.T_{wz}(P) + [x \neq z]\tilde{x}y.T_{wz}(P)$$

$$T_{wz}(x(y).P) = [x = z]w(y).T_{wz}(P) + [x \neq z]x(y).T_{wz}(P)$$

$$T_{wz}(P \backslash z') = T_{wz}(T(P \backslash z'))$$

**Proposition 2.2**: $P \simeq T(P)$ for any agent $P$.

**Proof**: Let $S = \{(P, T(P)) \mid P \text{ agent}\}$. Then $S$ is a strong bisimulation (up to $\alpha$-equivalence). For details see appendix.

We have thus proved the following:

**Theorem 2.6**: Blocking and mismatch have the same expressive power up to weak equivalence.

A point of interest should be noted here. In the proof of **Theorem 2.1** in [112], the following lemma is needed:

**Lemma**: If $P \xymatrix{\rightarrow \alpha} P'$, $\text{bn}(\alpha) \cap \text{fn}(P'\sigma) = \emptyset$, $\sigma[\text{bn}(\alpha)] = \text{id}$, then equally for some $P'' \equiv_{\alpha} P'\sigma$, $P\sigma \xymatrix{\rightarrow {\alpha}\sigma} P''$.

In our proof of the same lemma for $\pi B$, we must also require that $\sigma[\text{fn}(P)]$ be injective. The reason is that otherwise the lemma would be false. For instance, consider $(\pi x.0)\backslash b$ and $\sigma = \{b/a\}$. Without this restriction the lemma is also false for the $\pi$-calculus with mismatch for similar reasons. This gives further evidence to the fact that blocking and mismatch have the same expressive power.

### 3.3 Higher-Order $\pi$-calculus with Blocking

In the higher-order paradigm agents are allowed to be passed as values in a communication. In the standard $\pi$-calculus it was shown that reference passing is enough to represent process passing [149]. In the presence of dynamic binding as represented by the blocking operator this is no longer obvious, and thus we have to verify if the first-order paradigm is still powerful enough to express process-passing when blocking is present. Our main result in this section is that this is still possible by sending not a reference or a link to a process trigger as in [149], but by sending an encoding of the whole process, basically a representation of its parse tree. Nevertheless, this reduction is not complete, complex and rather awkward in practice.
We consider here the representability of the higher-order \( \pi \)-calculus extended with the blocking operator (from now on briefly \( \Pi B \)) within the first-order \( \pi \)-calculus extended with the blocking operator (from now on \( \pi B \)), and present an encoding of the reduced version of the higher-order \( \pi \)-calculus defined in [148] extended with blocking. This version includes only a subclass of the higher-order \( \pi \)-calculus, since only unary abstractions are allowed, and only one value, a name or a unary abstraction, can be exchanged in a communication. Moreover, we also impose that each agent in the calculus must be \textit{finitely} describable. Thus, sums, as well as the set of defined constants, are finite, making it possible to have replication instead of constants.

For \( \Pi B \) the issue of representability is much harder than for the higher-order \( \pi \)-calculus without blocking, since in \( \Pi B \) processes develop in some kind of “environment” where certain kinds of synchronisation are disallowed if the subject of the synchronisation occurs within the scope of some operator blocking it. In this case it is not enough to let the subject of a higher-order output action send a proxy or a link to a trigger for the process that is being exchanged in the synchronisation, instead of the process itself. For instance, we cannot represent

\[
\pi(P).Q | x(Y).(Y | Q')z
\]

as

\[
\pi(m).Q | mP | x(y).(m | Q')z
\]

because if \( P = \pi P' \) then after the triggering event there would be a possibility of communication along \( z \) in the encoding, whereas this possibility does not exist for the encoded agent.

Sending the name \( z \) instead of process \( P \) in the following fashion

\[
\pi m.(Q | m(z').(P' | z')) | x(y).(m | Q')z
\]

will also not do, since in the encoded agent we get after the first synchronisation

\[
Q | (P | Q')z
\]

and thus \( P \) may synchronise with \( Q' \) along the channel \( z \), whereas in the encoding the result would be

\[
(Q | P | Q')z | Q'z
\]

and here no synchronisation between \( P \) and \( Q' \) along \( z \) is possible.

### 3.3.1 Syntax

We consider here only the reduced version of the higher-order \( \pi \)-calculus defined by Sangiorgi in his thesis [148], extended with the blocking operator. In this subclass of agents only unary abstractions are allowed, and only one value, a name or a unary abstraction, can be exchanged in a communication. This restriction simplifies proofs and makes the results more readable. The reduced calculus may be easily generalised. Note that [148] adopted the scheme of early instantiation, which we adopt here as well.

Another restriction we impose is that each agent in the calculus must be \textit{finitely} describable. Thus, sums, as well as the set of defined constants, are finite, allowing us to have replication in place of constants. These restrictions are imposed in both \( \Pi B \) and \( \pi B \), since a certain uniformity in both calculi is desirable.
Agents are generated according to the following abstract syntax:

\[
P ::= 0 \mid \sum_{i=1}^{n} \alpha_i P_i \mid P_1 \mid P_2 \mid [x = y]P \mid (x)P \parallel P \parallel
\]
\[
P \parallel_2 \mid X(F) \mid X(x)
\]
\[
\alpha ::= \pi(F) \mid \pi y \mid x(X) \mid x(y)
\]
\[
F ::= (\lambda X)P \mid (\lambda x)P \parallel X
\]

Here \(x, y\) and \(z\) are metavariables over channel names, and \(X\) over agents. As before, we also use \(\pi y\) as a shorthand for \((y)\pi y\).

The higher-order nature of the calculus is brought out by the action prefix \(\alpha\). Output prefix actions (of the shape \(\pi y\) or \(\pi (F)\)) can pass names as well as general agent abstractions \(F\).

Restrictions \((x)P\), abstractions \((\lambda x)P\) and \((\lambda X)P\), as well as input action prefixes of type \(a(x)P\) or \(a(X).P\), bind free occurrences of \(x\) and \(X\) in \(P\). There are no other operators with binding power. Terms are identified up to \(\alpha\)-conversion. We let \(n(P)\) represent the set of names occurring in the agent \(P\), \(fn(P)\) the set of free names in \(P\), and \(bn(P)\) the set of bound names in \(P\).

**Sorting**

We assume agents in \(\Pi B\) are well-sorted according to the following sorting discipline. Subject sorts and object sorts are called element sorts, and the symbol \(El\) is used to range over element sorts.

\[
El ::= s \mid S
\]
\[
S ::= () \mid (El)
\]

For each object sort \(S\) we assume the existence of an infinite number of variables of sort \(S\), \(Var(S)\).

The set of formation rules for well sorted agents of \(\Pi B\) is presented below. Names are supposed to be partitioned in a collection of subject sorts. That \(x\) belongs to subject sort \(s\) is written \(x : s\). Object sorts are defined recursively as either the empty object sort \(()\), or \((El)\) for some element sort \(El\). A sorting is a function \(Ob\) mapping subject to object sorts. Also, \(a : s \mapsto (El)\) means that the name \(a\) belongs to the subject sort \(s\) and that the object sort of \(s\) is \((El)\). An agent \(A\) is well-sorted if we can infer \(A : S\) for some \(S\) from the rules in the table below. If \(S = (El)\) then \(A\) is an abstraction, whereas if \(S = ()\) then \(A\) is a process.

Below, \(K\) stands for an abstraction or a name, and \(U\) for a variable or a name.
### Formation Rules for LΠB

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X \in \text{Var}(S) \quad X : S )</td>
<td>( A : () \quad !A : () )</td>
</tr>
<tr>
<td>( U : El \quad A : () )</td>
<td>( \lambda U \cdot A : { El } )</td>
</tr>
<tr>
<td>( K : El \quad A : () \quad \alpha : s \rightarrow { El } )</td>
<td>( \alpha(K) \cdot A : () )</td>
</tr>
<tr>
<td>( U : El \quad A : () \quad \alpha : s \rightarrow { El } )</td>
<td>( \forall \xi \cdot A_{\xi} : () \quad \sum_{\xi \in El} A_{\xi} : () )</td>
</tr>
<tr>
<td>( A_1 : () \quad A_2 : () )</td>
<td>( A : () \quad { x } A : () )</td>
</tr>
<tr>
<td>( x : y \quad A : () )</td>
<td>( [x = y] A : () \quad X : { El } \quad K : El )</td>
</tr>
<tr>
<td>( X(K) : () )</td>
<td>( A : () )</td>
</tr>
</tbody>
</table>

| \( A \setminus \{ x \} \) | \( A : () \) |

#### 3.3.2 Operational Semantics

There are three kinds of actions \( \mu \): the silent action, input action and output action. Bound names and free names of an action \( \mu \), \( \text{bn}(\mu) \) and \( \text{fn}(\mu) \) resp, are defined as follows:

<table>
<thead>
<tr>
<th>Action</th>
<th>Bound Names</th>
<th>Free Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x(y) )</td>
<td>( { x, y } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( (y) \setminus x )</td>
<td>( { x } \cup (\text{fn}(K) - y) )</td>
<td>( y )</td>
</tr>
<tr>
<td>( \tau )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Note that \( y \) in \( x(y) \) is not bound, in accordance with the early instantiation scheme adopted here.

The names of \( \mu \), briefly \( n(\mu) \), are defined as \( \text{bn}(\mu) \cup \text{fn}(\mu) \); \( s(\mu) \) is the singleton set containing the subject \( x \) of an action; \( \tilde{y} \) represents a tuple of names, and \( (\tilde{y}) \) is shorthand for \( (y_1 \ldots y_n) \) whenever \( \tilde{y} = (y_1 \ldots y_n) \). We use also \( \tilde{y} \) ambiguously as a shorthand for the set of names occurring in the vector \( \tilde{y} \).

The action rules for LΠB are the following:
### Rules of Action for II$	ext{B}$

| ALP: $P \xrightarrow{\mu} P'$, $P$ and $P'$ are $\alpha$-convertible |
| OUT: $\pi \{K\}. P \xrightarrow{\pi \{K\}} P$ |
| INP: $x[U]. P \xrightarrow{\pi \{K\}} P[K/U]$, if $K$ and $U$ have the same sort |
| SUM: $\sum_{i=1}^{n} P_i \xrightarrow{\mu} P'$, $1 \leq k \leq n$ |
| PAR: $P \xrightarrow{\mu} P'$, $P \parallel Q \xrightarrow{\mu} P' \parallel Q$, $\text{bn}(\mu) \cap \text{fn}(Q) = \emptyset$ |
| COM: $P \xrightarrow{\mu} P', Q \xrightarrow{\pi \{K\}} Q'$ |
| MATCH: $P \xrightarrow{\mu} P'$, $[x = x] P' \xrightarrow{\mu} P'$ |
| REP: $P \parallel P' \xrightarrow{\mu} P' \parallel P'$ |
| RES: $P \xrightarrow{\mu} P'$, $[x] P \xrightarrow{\mu} [x] P'$ |
| OPEN: $P \xrightarrow{\pi \{K\}} P'$, $x \notin \mu$, $x \in \text{fn}(K) - \text{fn}(Q)$ |
| BLOCK: $P \xrightarrow{\mu} P'$, $z \notin \mu$ |

**Obs:** Symmetric forms for operators $+$ and $-$ have been omitted

### 3.3.3 The Reduction

In order to represent II$	ext{B}$ in $\pi B$, we apply a transformation $\mathcal{T}$, a function from II$	ext{B}$ to $\pi B$. We will show that $P$ and $\mathcal{T}(P)$ are weakly equivalent in a sense that will be defined in Section 3.3.4, if $P$ is closed and well-sorted.

Some channels are associated with the operators and should not be used for other purposes. These are: $z, c, s, m, r, b, i$. They represent the process $\theta(z)$, composition ($c$), sum ($s$), matching ($m$), replication ($r$), blocking ($b$), input ($i$) and output ($o$).

We assume that for each name $x$ in II$	ext{B}$ there corresponds a unique name $x$ in $\pi B$, and also that for each process variable $Y$ in II$	ext{B}$ there corresponds a unique channel $y$ in the target calculus $\pi B$.

The basic idea is that pointers to abstractions, instead of abstractions themselves, are objects of communication. To each abstraction $(\lambda X)P$ that is the object of a communication there corresponds a spawning process $\text{spawn}_m(\mathcal{F})$. This process can continuously receive pointers $y$ to abstractions instantiating $X$, upon which it launches a process of type $\text{send}_m(P\{Y/X\})$, whose task is the transmission of an encoding of $P\{Y/X\}$. Concurrently, a process of type $\text{rec}(\mathcal{W})$, which we call receiver or interpreter, receives and interprets the encoding of a process $P\{Y/X\}$, emulating it dynamically. Receiver processes arise in connection with applications of type $Y(\mathcal{F})$, in which $Y$ is not instantiated directly by an abstraction, but by a pointer to an abstraction, and must emulate the behaviour of $Y(\mathcal{F})$. 

40
The three agents $spawn_w(F)$, $send_v(P)$ and $rec(v)$ form the the core of the transformation $T$, and are explained in detail below.

**Higher-Order Input** For the input operator the transformation is defined as follows:

$$T(x(Y).P) = x(y).T(P).$$

The idea is that instead of receiving a process $Q$ that instantiates the process variable $Y$ in $P$, thus becoming $P\{Q/Y\}$, a “pointer” $y$ to $Q$ is received.

**Higher-Order Output** In the central case of higher-order output we get:

$$T(\pi(F), P) = \pi(w).\pi(y).T(P).$$

provided $F$ is not a process variable. Here, instead of communicating the abstraction $F$, a “pointer” $w$ to $F$, or rather to a process responsible for spawning encodings of $F$, $spawn_w(F)\|$, is sent instead. If the abstraction $F$ is a process variable $Y$, what is communicated is its corresponding pointer $y$:

$$T(\pi(Y), P) = \pi(y).T(P).$$

For closed processes, this situation can arise only if $Y$ occurs within the scope of some input prefix $a(Y)$ for some channel $a$, for instance $a(Y), \pi(Y).P$. In this case $Y$ must have been previously instantiated to some agent $F$ and a spawning process for $F$, $spawn_y(F)$, must have been launched elsewhere.

**Application** Since we are dealing only with closed processes, an application $Y(F)$ must occur within the scope of some input prefix $a(Y)$ for some channel $a$, and thus $Y$ must have been previously instantiated to some agent $G$ and a process of type $spawn_y(G)$ must have been launched elsewhere and be running in parallel with $T(Y(F))$, which reduces to the following:

$$T(Y(F)) = \gamma(u).\pi(v).\pi(u).rec(v) | spawn_y(F).$$

The application of the process $G$ to its argument $F$ is executed by the “receiver” process $rec(v)$, an agent whose function is to enact a copy of $G(F)$ in the environment where it occurs (possibly within the scope of some blocking operators) by means of the reception and execution of an encoding of $G(F)$ along $v$, a fresh channel sent to the spawning process $spawn_y(G)$ along channel $u$ for this purpose. The task of sending an encoding of a process is performed by a sender process which needs to know the pointer $w$ to the agent $F$ being applied. For this purpose the $w$ is also communicated to $spawn_y(G)$.

If the argument of $Y$ is a name $x$, then this name is communicated to the spawning process $spawn_y(G)$:

$$T(Y(x)) = \gamma(u).\pi(v).\pi(u).rec(v).$$

If the argument is a process variable $X$ a similar construction is used.

As an example, the higher-order process

$$\pi((\lambda X)P).Q | x(Y).Y(F)$$

reduces to the first-order process

$$\pi(w).\pi((T(Q) | w(u).u(v).u(x).send_v(P))). | x(y).\gamma(u).\pi(v).\pi(u').rec(v) | spawn_w(F)).$$

The agent $send_v$ is explained below.
Senders

The process $\text{spawn}_w(F)$, assuming $F = (\lambda X)P$ or $(\lambda x)P$, spawns continuously, for any received $v$, “sender” processes of type $\text{send}_v(P)$, whose task is the transmission along $v$ of encodings of $P$ with $X$ or $x$ instantiated to a pointer to the process instantiating $X$ resp a channel instantiating $x$:

$$\text{spawn}_w((\lambda x)P) = w(u(v).u(x)\text{send}_v(P))$$

$$\text{spawn}_w((\lambda Y)P) = w(u(v)u(y)\text{send}_v(P))$$

In order to perform its task, the sender process $\text{send}_v(P)$ must make use of special channels indicating the nature of $P$’s head operator, and which should not be used for other purposes.

Null agent  In case $P = 0$, an indication of this will be provided through the action prefix $\overline{\nu}z$, upon which the “sender” $\text{send}_v(P)$ terminates. The name $z$ indicates here the null process.

Matching For matching the sender process need only provide both elements of the equality:

$$\text{send}_v([x = y](P)) = \overline{\nu}m.\overline{\nu}x.\overline{\nu}y.\text{send}_v(P)$$

Restriction Restriction is defined thus:

$$\text{send}_v((x)P) = (x)\text{send}_v(P).$$

Nothing needs be communicated here. If $x$ is to be extruded along some channel, say $y$, the sender communicates $y$ and $x$ to the receiver, which will then offer $\overline{\nu}x$ for synchronisation, in this way mimicking exactly the original extrusion procedure.

Input Communicating an input is only slightly more complicated. In this case, the channel along which the input occurs is communicated to the receiver, which is supposed to dynamically enact such input synchronisation before sending back to the sender the actual parameter, which is the name exchanged in the synchronisation. The sender will thus wait for the communication of this channel which instantiates $y$, whereupon it goes on sending an encoding of the continuation of the prefixed agent:

$$\text{send}_v(x(y).P) = \overline{\nu}i.\overline{\nu}x.\overline{\nu}y.\text{send}_v(P)$$

No distinction is made for higher-order inputs, since in this case what is communicated by the sender is a “pointer” to a process, which is also a channel.

Output The encoding of an output action involving only variables is straightforward. Both the subject and the object of the action are communicated, if the latter is a process variable its corresponding channel is provided:

$$\text{send}_v(\overline{\nu}y.P) = \overline{\nu}o.\overline{\nu}x.\overline{\nu}y.\text{send}_v(P).$$

$$\text{send}_v(\overline{\nu}Y.P) = \overline{\nu}o.\overline{\nu}x.\overline{\nu}y.\text{send}_v(P).$$

In the case of a higher-order output $\overline{\nu}(F).P$ involving an abstraction $F$, a fresh channel $v$, a “pointer” to the abstraction, is provided by the sender, whereupon a spawning process for $F$, $\text{spawn}_w(F)$, is initiated in parallel with $\text{send}_v(P)$:

$$\text{send}_v(\overline{\nu}(F).P) = \overline{\nu}o.\overline{\nu}x.\overline{\nu}(w).\text{send}_v(P) \; \mid \; \text{spawn}_w(F))$$
Blocking  _Blocking_ is straightforward: the channel being blocked is communicated and then the encoding of the process within the scope of the blocking operator:

\[
send_u(P|z) = \forall v. \forall z. send_u(P)
\]

Parallel Composition  If \( P = P_1 \mid P_2 \), then \( c \), representing composition, is communicated along \( v \), followed by the exchange of a couple of fresh pointers to both components of \( P \), \( P_1 \) and \( P_2 \), upon which two new sender processes are created that provide an encoding of \( P_1 \) resp. \( P_2 \):

\[
send_u(P_1 \mid P_2) = \forall c. \forall (v_1). \forall (v_2). (send_{u_1}(P_1) \mid send_{u_2}(P_2)).
\]

Replication  For replication \( !P \) the sender continuously communicates as many copies of \( P \) as requested, previous communication of a fresh channel for each new copy of \( P \):

\[
send_u(!P) = \forall v. \forall (v'). send_u(P)
\]

Summation  The definition of \( send_u(\sum_{i=1}^{n} P_i) \), \( n > 1 \), includes the exchange of a couple of fresh “pointers”, \( v_i \) for \( P_1 \) and \( v_2 \) for \( \sum_{i=2}^{n} P_i \), which in case \( n = 2 \) must be a prefixed agent. This scheme works because only well-guarded agents are allowed in summations.

Application  _Application_ \((Y\langle x \rangle), Y\langle X \rangle\) or \( Y\langle F \rangle\) presents a special subtlety: this is the only case where the sender process does not communicate directly with the corresponding receiver, but instead with the spawning process \( spawn_u(F) \) associated with the process variable \( Y \), which at this time must have been instantiated since we are considering only closed agents. The sender simply communicates, along channel \( y \), the channel \( v \) along which it is presently synchronising with its corresponding receiver \( rec(v) \), as well as the argument of the application or a pointer to it. If the argument is only a channel, \( x \), then this channel is communicated; if it is a process variable \( X \), then, since for some agent \( G \) this variable must have been instantiated to some channel acting as a “pointer” to a spawning process \( spawn_u(G) \) associated with \( X \), the channel \( x \) is also provided; otherwise, if the argument is an abstraction \( F \), a fresh pointer \( w \) to a new spawning process \( spawn_w(F) \) is communicated, whereupon \( spawn_w(F) \) is initiated. In the first two cases there is no need to start a new spawning process, and therefore the sender is simply terminated, since the receiver \( rec(v) \) goes on to synchronise with the sender process launched by \( spawn_w(F) \) in a transparent way:

\[
send_u(Y\langle x \rangle) = \forall y. \forall v. \forall x. 0
\]

\[
send_u(Y\langle X \rangle) = \forall y. \forall v. \forall x. 0
\]

\[
send_u(Y\langle F \rangle) = \forall y. \forall v. \forall w. spawn_w(F)
\]

Receivers

The task of the receiver or interpreter \( rec(v) \) is to receive from a sender \( send_u(P) \), along the channel \( v \), the encoding of a process \( P \), and at the same time to interpret this encoding.

Null agent  If the process being communicated is \( 0 \), then the receiver simply enacts it as \( 0 \):

\[
[u = z] 0
\]
**Matching** For *matching*, the receiver, after reception of indication that the head operator of the process being encoding is matching, receives both names being matched, say $x$ and $y$, becoming thus $[x = y] \text{rec}(v)$:

\[ [u = m] v(x). v(y). [x = y] \text{rec}(v). \]

**Input** For input the task of the receiver is to emulate any of these actions by dynamically offering the subject $x$ of the action for communication. The receiver offers a synchronisation along $x$, upon which it communicates to the sender the channel exchanged in the synchronisation. In case of higher-order agent input the the situation is the same as for channel input, since only pointers to agents are involved here:

\[ [u = i] v(x). x(y). \exists y. \text{rec}(v). \]

**Output** This case is similar. If the action is a first-order output, then the same action is carried out by the receiver. If it is a higher-order output, then what is communicated is the “pointer” to the spawning process associated with the process that is the object of the action. In both cases the behaviour of the receiver is the same:

\[ [u = o] v(x). v(y). \exists y. \text{rec}(v). \]

**Blocking** The treatment of *blocking* is straightforward: the receiver simply applies this operator to itself:

\[ [u = b] v(z). (\text{rec}(v)) \setminus z. \]

**Parallel Composition** For process composition $P = P_1 \parallel P_2$, the receiver requires a pair of fresh pointers for each of $P_1$ and $P_2$, whereupon it gives rise to a composition of two new receiver processes, $\text{rec}(v_1) \parallel \text{rec}(v_2)$, whose task is to receive an encoding of $P_1$ resp $P_2$ and execute them:

\[ [u = c] v(v_1). v(v_2). (\text{rec}(v_1) \parallel \text{rec}(v_2)). \]

**Replication** *Replication* may be treated here as a generalisation of composition. Whenever a copy of the replicated process is activated, a new fresh pointer to this new copy is exchanged, and a new version of the receiver synchronising along this fresh channel is activated:

\[ [u = r] v(v'). \text{rec}(v'). \]

**Summation** The most difficult part of the receiver, and illustrating the difficulties in extending generality beyond static operators, is the encoding of summation. We handle this using semaphores and backtracking (see also [124] for a similar approach of encoding choice). Assume that the process being encoded is $\sum_{i=1}^{n} \alpha_i.P_i$. After having received indication that the next head operator is summation, the receiver initiates the following process:

\[ (a)(p)(q)(U(vuapq) \mid a.Sem(pq)). \]

The process $Sem(pq)$ is a semaphore, and may be initiated only after synchronisation through channel $a$ acting as a guard. This synchronisation is executed together with the process $U(vuapq)$ and may happen only after the latter has given rise to a series of process running in parallel and mimicking $\sum_{i=1}^{n} \alpha_i.P_i$. The process $U(vuapq)$ gets an indication of the kind of action, input or output, that each summand of the well-guarded summation is ready to accept. If the first
The Reduction

The definition of the transformation \( \mathcal{T} \) is as follows:

\[
\begin{align*}
\mathcal{T}(0) & = 0 \\
\mathcal{T}(xy.P) & = xy.\mathcal{T}(P) \\
\mathcal{T}(x(y).P) & = x(y).\mathcal{T}(P) \\
\mathcal{T}(P_1 \mid P_2) & = \mathcal{T}(P_1) \mid \mathcal{T}(P_2) \\
\mathcal{T}(\sum_{i=1}^n P_i) & = \sum_{i=1}^n \mathcal{T}(P_i) \\
\mathcal{T}( [x = y].P) & = [x = y].\mathcal{T}(P) \\
\mathcal{T}( (x).P) & = (x).\mathcal{T}(P) \\
\mathcal{T}( !P) & = !\mathcal{T}(P) \\
\mathcal{T}(P \setminus z) & = \mathcal{T}(P) \setminus z \\
\mathcal{T}( \pi(Y).P) & = \pi y.\mathcal{T}(P) \\
\mathcal{T}( \pi(F).P) & = \pi w.(\mathcal{T}(P) \mid \text{spawn}_w(F)), \text{ F not a process variable} \\
\mathcal{T}(xY).P & = x(y).\mathcal{T}(P) \\
\mathcal{T}(Y(x)) & = \gamma(u).\pi v.(\pi x.\text{rec}(v)) \\
\mathcal{T}(Y(X)) & = \gamma(u).\pi v.(\pi x.\text{rec}(v)) \\
\mathcal{T}(Y(F)) & = \gamma(u).\pi v.(\pi w.(\text{rec}(v) \mid \text{spawn}_w(F))), \\
\text{ F not a process variable}
\end{align*}
\]

\[
\begin{align*}
\text{spawn}_w((\lambda x).P) & = !w(u).u(v).u(x).\text{send}_v(P) \\
\text{spawn}_w((\lambda Y).P) & = !w(u).u(v).u(y).\text{send}_v(P)
\end{align*}
\]
\[
\begin{align*}
\text{send}_v(0) &= \pi_z 0 \\
\text{send}_v(P_1 | P_2) &= \pi v. (\text{send}_v(v_1) | \text{send}_v(P_1)) \\
\text{send}_v(\sum_{i=1}^n P_i) &= \pi v. (\text{send}_v(v_1) | \text{send}_v(P_1)) \\
\text{send}_v([x = y](P)) &= \pi v. \text{send}_v(P) \\
\text{send}_v((x)P) &= (x) \text{send}_v(P) \\
\text{send}_v((P)z) &= \pi v. \text{send}_v(P) \\
\text{send}_v(x(y).P) &= \pi i. \text{send}_v(P) \\
\text{send}_v((x Y).P) &= \pi i. \text{send}_v(P) \\
\text{send}_v((Y x).P) &= \pi v. \text{send}_v(P) \\
\text{send}_v((Y F).P) &= \pi v. \text{send}_v(P) \\
\text{send}_v((Y (F)).P) &= \pi v. \text{send}_v(P) \\
\text{rec} &\overset{\text{def}}{=} (v) v(u).
\end{align*}
\]

\[
\begin{align*}
| u = z 0 |
\end{align*}
\]

\[
\begin{align*}
| u = c v(v_1.0) v(v_2) (\text{send}(v_1) | R(v_2)) |
\end{align*}
\]

\[
\begin{align*}
| u = s i (p)(q)(U(v u a p q) | a. \text{Sem}(p q)) |
\end{align*}
\]

\[
\begin{align*}
| u = r v x. v(y). [x = y] \text{rec}(v) |
\end{align*}
\]

\[
\begin{align*}
| u = b v(z). (\text{rec}(v)) y z |
\end{align*}
\]

\[
\begin{align*}
| u = o v x. v(y). \pi x y. \text{rec}(v) |
\end{align*}
\]

\[
\begin{align*}
| u = i v x. v(y). \pi x y. \text{rec}(v) |
\end{align*}
\]

\[
\begin{align*}
U &\overset{\text{def}}{=} (v u a p q) \\
| u = s i v(v_1.1) v(v_2.2) v(v_1.3) |
\end{align*}
\]

\[
\begin{align*}
| u = i v x. v(y). (U(v_1.4) | U(v_2.5) | U(v_3.6)) |
\end{align*}
\]

\[
\begin{align*}
| u = i v x. \pi I (v x p q) |
\end{align*}
\]

\[
\begin{align*}
| u = o v x. v(y). \pi O (v x y p q) |
\end{align*}
\]

\[
\begin{align*}
I &\overset{\text{def}}{=} (v x p q) \pi (x(y) \pi y. \text{rec}(v) + q I (v x p q)) \\
O &\overset{\text{def}}{=} (v x y p q) \pi (x(y) \pi y. \text{rec}(v) + q O (v x y p q)) \\
\text{Sem} &\overset{\text{def}}{=} (pq) p q. \text{Sem}(pq)
\end{align*}
\]

3.3.4 Barbed Bisimulation

The issue of the representability of $\Pi B$ within $\pi B$ requires both the choice of a precise semantics for each of these calculi, and the definition of some kind of operational relation between these semantics, in order to make it possible to give a
clear definition of the notion of encoding correctness. The encoding should be compositional in the sense that the encoding of an agent should depend only on the encoding of its components, a requirement fulfilled by the transformation $T$ defined above.

In accordance with the predominant approach to the semantics of concurrent systems, we base our definition of encoding correctness on the operational behaviour. The most straightforward way to do it is by defining some kind of behavioral equivalence that is uniform in both calculi, and therefore we decided to base our approach on the notion of barbed bisimulation [113]. For higher-order agents this is a convenient approach, as it permits us to direct primary attention at the channels along which communication takes place, rather than the parameters.

However, blocking does not allow any simple formulation of structural congruence in the spirit of the Chemical Abstract Machine of Berry and Boudol [21]. The reason is that blocking cannot be distributed over the composition operator even if the channel being blocked does not occur in some of the operands. For instance, even if $x \neq a$ and $x \neq b$, it is false that

$$((a \cdot b) \cdot (b \cdot z) \cdot 0 | \pi y, Q) \cdot \pi \equiv (a \cdot b) \cdot (b \cdot z) \cdot 0 | \pi y, Q \cdot \pi \sim (a \cdot b) \cdot (b \cdot z) \cdot 0$$

since if $b$ in $(a \cdot b) \cdot (b \cdot z) \cdot 0$ is instantiated to $x$ then

$$((a \cdot b) \cdot (b \cdot z) \cdot 0 | \pi y, Q) \cdot \pi \xrightarrow{a(x)} (x \cdot z) \cdot 0 | \pi y, Q \cdot \pi \xrightarrow{T} (0 | Q) \cdot \pi \equiv Q \cdot \pi$$

This reflects the fact that blocking enhances local control, and necessarily affects the structure of interactive systems. In a sense blocking reflects the structure induced by the linkage among system components and represents spatial or static structure better than static restriction, which may always be pushed outside of expressions through renaming.

Full abstraction, that is, the requirement that two terms in $\Pi B$ be equivalent if and only if their translations in $\pi B$ are equivalent, is not fulfilled by the translation $\mathcal{T}$. The reason is, roughly, that sending a process $P$ is like sending the object code of a program, the receiver may only enact it but not change it, whereas sending an encoding of $P$ is like sending the source code, the receiver may change the program at will and also its own behaviour in accordance with the nature of any of the components of $P$. For instance, for any process $P \in \Pi B$, $\pi((\lambda x) P) \cdot 0$ and $\pi((\lambda x) P | 0) \cdot 0$ are certainly equivalent. Nevertheless, their translations are quite distinct. In the former case an agent $\text{send}_a(P)$ will eventually be activated, whereas in the latter case the agent activated will be $\text{send}_b(P | 0)$. The latter provides an encoding of $P | 0$, not $P$, and it does it by first sending an indication that the head operator is the composition operator, $\pi c$. Any process in $\pi B$ synchronising with $\text{send}_a(P | 0)$ may choose to act according to the nature of this synchronisation, for example $w(x) \cdot (x = c) \cdot 0 + (x = e) \cdot (Q)$. Thus, the translations of $\pi((\lambda x) P) \cdot 0$ and $\pi((\lambda x) P | 0) \cdot 0$ cannot possibly be equivalent in any sensible way. Nevertheless, a restricted form of completeness is achieved by the translation $\mathcal{T}$ if we limit testing on terms in $\pi B$ to encodings of source terms. In this restricted form the translation proposed here is both sound and complete.

Since full abstraction may be achieved in a variety of ways, some kind of operational correspondence between a term and its translation should also be given. This is done here by a bisimulation involving pair of processes of form $(P, T(P))$, where $P \in \Pi B$. What makes this possible is the fact that the semantics of $\pi B$ and $\Pi B$ are uniform.

Before defining the distinct forms of barbed bisimulation, we need the following:
**Definition 3.1 (Static Context):** A static context $C[\cdot] \in \pi B$ ($\Pi B$) is any element of the set generated by the following grammar, where $P$ ranges over the agent language under consideration.

\[
C ::= P | [] | C| C (x)C | \epsilon C | C\epsilon
\]

A static context $C[\cdot]$ may be viewed as an agent with a “hole” $[]$ in it, and $C[P]$ the result of substituting $P$ for every occurrence of $[]$ in $C$.

Let $P \Downarrow_a$ hold just in case $P \xrightarrow{\alpha} Q$ for some $Q$ and $\alpha$ such that $s(\alpha) = \{a\}$. Let also $\xrightarrow{\cdot}$ be defined as $\xrightarrow{\cdot}$, and $\xrightarrow{\cdot} \cdot$ as the reflexive and transitive closure of $\xrightarrow{\cdot}$.

Let $P \Downarrow_a$ mean that $P \Rightarrow P' \Downarrow_a$ for some $P'$.

**Definition 3.2 (Barbed Bisimulation):** A binary relation $R$ on processes is a strong (weak) barbed simulation if $PQ \Rightarrow RQ'$ implies:

1. Whenever $P \xrightarrow{\cdot} P'$ then $Q \xrightarrow{\cdot} Q'$ ($Q \Rightarrow RQ'$) for some $Q'$ such that $P' \Rightarrow RQ'$.
2. For each $a$, if $P \Downarrow_a$ then $Q \Downarrow_a (Q \Downarrow_a)$.

A relation $R$ is a barbed bisimulation if $R \Rightarrow R^{-1}$ are barbed simulations.

The agents $P$ and $Q$ are strong (weak) barbed-similar, written $P \leadsto Q (P \leadsto Q)$, if $PSQ$ for some strong (weak) barbed bisimulation $S$. If the agents $P$ and $Q$ belong to the same language, then they are strong (weak) barbed equivalent, written $P \leadsto Q (P \leadsto Q)$, if for each static context $C[\cdot]$ in the language it holds that $C[P] \leadsto C[Q]$ ($C[P] \leadsto C[Q]$).

### 3.3.5 Soundness and Completeness of Reduction

Before stating the main result of this chapter, we need the following.

**Definition 3.3 (Reduced Composition, Reduced Context):**

1. A reduced composition $\prod$ is a composition in $\pi B$ of agents of type $\text{spawn}_w(F)$, $\text{send}_v(P)$, $\text{rec}(v)$, $T(P)$, or any of the derivatives of such agents, for any agents $F$ and $P \in \Pi B$, and such that (i) if $\text{spawn}_w(F)$ and $\text{spawn}_w(G)$ occur in $\prod$, and $w = w'$, then $F \equiv G$; (ii) if $\text{send}_v(P)$ and $\text{send}_v(Q)$ occur in $\prod$, and $v = v'$, then $P \equiv Q$.
2. A context $C[\cdot] \in \pi B$ is called a reduced context if $C[\cdot] = (\bar{y})[\prod | []]$ for some channel vector $\bar{y}$ and some reduced composition $\prod$ with no occurrence of the restriction operator.

**Definition 3.4 (Reduced Equivalence):** Two processes $P$ and $Q \in \pi B$ are strong (weak) reduced equivalent, written $P \leadsto_r Q (P \leadsto_r Q)$, if for each reduced context $C[\cdot] \in \pi B$, it holds that $C[P] \leadsto C[Q]$ ($C[P] \leadsto C[Q]$).

Reduced equivalence is an equivalence relation. Moreover, from the definition it is immediate that for any processes $P$ and $Q$ in $\pi B$, $P \leadsto Q$ implies $P \leadsto_r Q$, and $P \leadsto Q$ implies $P \leadsto_r Q$.

Strong and weak reduced equivalence are congruences under output prefix, bang ($\cdot$), restriction, and blocking. For $| \bullet$ let $P \leadsto_r Q$ and $P' \leadsto_r Q'$ such that $P, P', Q, Q'$ are in the range of $T$. If $\prod$ is a reduced composition, then so is $\prod' = \prod | P$ for any
$P$ in the range of $\mathcal{T}$. Thus we obtain, assuming channels have been conveniently renamed and all occurrences of restrictions have been pushed outside:

$$(\bar{y})(\prod | P \mid P') \sim (\bar{y})(\prod | P \mid Q') \quad \text{(since } P' \sim_r Q')$$

$$\sim (\bar{y})(\prod | Q' \mid P)$$

$$\sim (\bar{y})(\prod | Q' \mid Q) \quad \text{(since } P \sim_r Q)$$

$$\sim (\bar{y})(\prod | Q \mid Q').$$

The same relation holds for the weak case.

We sum up this in the following proposition:

**Proposition 3.1** Strong and weak reduced equivalence are congruences under output prefix, bang ($!$), restriction, and blocking.

The result by which we prove correctness is the following:

**Theorem 3.1:** $P \approx Q \iff \mathcal{T}(P) \approx_r \mathcal{T}(Q)$, i.e. $\mathcal{T}$ is both sound and complete with respect to weak barbed equivalence in $\Pi B$ and weak reduced equivalence in $\pi B$.

A detailed proof of this theorem is given in the Appendix A.2.

### 3.4 Conclusions

Blocking enhances local control, reflects the structure induced by the linkage among system components and seems to capture the idea of location adequately. In the first-order context blocking may be expressed by match and mismatch, but in the higher-order calculus the matter is much more complex. We offered a reduction that is not fully abstract, which is not without consequences, as explained below. Completeness is achieved only by limiting testing on target terms to encodings of source terms.

In the security protocol application that initiated our study little is assumed about the behaviour of an eventual intruder, and it is by no means obvious how we may formalise security features such as cryptographic keys or the impossibility of an intruder to obtain confident information. Nevertheless, a reduction of a higher-order representation of a cryptographic protocol to first-order, using the method we described in the previous section, requires some kind of “well-behaviour” on the part of the intruder in the first-order setting, for instance that it must behave in a way that may be described as a reduction of some kind of behaviour definable in the higher-order calculus. Thus, a formalisation of a security feature in the higher-order setting may be useless after reduction to the first-order setting. For instance, in the target calculus both keys and messages must be passed separately along some channel, instead of the whole agent representing the encrypted message, and thus snatching a message is equivalent with snatching its contents, in contrast with snatching a process, which does not necessarily give immediate access to its structure.

The focal point is: is the justification for having process passing in a calculus of communicating systems limited to such features as the power and elegance of its abstractions techniques, the fact that many systems may be more easily or naturally described in a higher-order setting, or is it more than that? It could
very well be the case that there is a more suitable reduction strategy that we have not been able to find. Nevertheless, other researchers ([5], [4]) were also forced to extend the π-calculus in order to model cryptographic features, which strengthens the view that there is no obvious solution to the problem. This fact seems to give further evidence to our claim that first-order passing is not suitable for emulating the exchange of confidential information, unless it is done along restricted channels, which is begging the question since the crucial point in cryptographic protocols is that confidential information must be exchanged along an insecure network. What distinguishes higher-order process-passing is the fact that the receiver of a process does not necessarily gain any knowledge about the channels that occur in the received process, a fact that permits us to represent encrypted information as a special kind of process. This does not happen with first-order channel-passing, since the receptor of a channel will immediately get complete knowledge about it and is thus always able to use it as a means of communication. In contrast, one may acquire a process through communication without necessarily being able to get any knowledge about its structure. This is a suitable way of representing encrypted information, which may be viewed, in accordance with basic principles in process calculi, as a kind of active agent that is entitled to yield further information to anyone in possession of some required previous information, e.g. cryptographic keys.

Nevertheless, cryptographic protocols are by nature very subtle and methods for studying higher-order process calculi seem to be unduly complex for the application in question, which only requires one special kind of process in process-passing and one kind of public channel. Techniques and tools that have been developed for the full calculus may turn out to be unmanageable for the study of cryptographic features.

Finally, any operator that provides local control of communication in a higher-order setting is likely, as the blocking operator, to interact badly with Sangiorgi’s basic result [149] showing that higher-order features in the π-calculus are reducible to first-order ones. We have resolved this by providing a very general higher-order reduction, based on the idea of communicating and dynamically interpreting parse trees in place of the processes themselves. We conjecture that any “reasonable” static operator can be handled in this way. It would be interesting to prove such a statement in terms of an extension of one of the well-known formats for operational semantics such as GSOS [23], adapted to the π-calculus.
Chapter 4

Blocking and Filtering with Polarised Channels

There are many occasions in which it is convenient to impose some constraint on the behaviour of a process with regard to its communication capabilities. Examples include resource access sharing and communication of capabilities to a process. The idea is to avoid interference or undesirable communication along a particular channel. Usually, this kind of constraint is enforced by imposing a type discipline on channels. Processes are defined as well-sorted or well-typed in case their bound and free names meet some particular set of conditions delimiting the communication capabilities along these names ([170, 134, 150, 54]). However, within the context of open systems this scheme is inadequate, since such systems are assumed to consist of an extensible number of arbitrary agents that do not necessarily follow any pre-determined protocol. An alternative, that we shall explore in depth here, is to add primitives expressing limitation of the communication capabilities of an agent.

In this chapter we are going to investigate how notations based on the $\pi$-calculus may be extended with primitive operators that restrict communication capabilities. Basically, we extend the $\pi$-calculus with two new operators, which we call polarised blocking resp. polarised filter, or simply blocking and filter. Polarised blocking is a refinement of the blocking operator introduced earlier. The difference with plain blocking is that polarised blocking may block communication along one direction only. Hence, output actions along a certain channel may be blocked, but not input actions, and vice versa. Filtering, on the other hand, may be seen as a kind of dual to blocking. A process filtered on a set of polarised channels, the filter set, is able to communicate with its environment only if the action is allowed by the filter set, i.e. only if the channel along which communication takes place, together with the corresponding polarity, belongs to the filter set. Hence, a process within the scope of a filter operator and a filter set can be seen as a process blocked on every polarised channel that does not belong to the filter set.

The usefulness of polarised blocking and filtering will be illustrated by showing how they may be applied to cases involving resource access sharing, access rights, and protection domains. We contrast our approach to one where behaviour restriction is based on the notion of capability defined in terms of types. Furthermore, we will show how restrictions on the behaviour of processes that are based on type systems may be defined in terms of blocking and filtering. To this purpose, we consider in detail a typed higher-order calculus based on the $\lambda$ and the $\pi$-calculus, the $\lambda\pi_v$ calculus, recently presented by Hennessy and Yoshida [187]. This calculus is the first one, to our knowledge, where types are given to a process specifying the
interface of the process with the purpose of delimiting its access rights. We will show that access rights may alternatively be enforced by restricting the interface of a process by means of either the filter or the blocking operator. We develop a new calculus based on these principles, similar to the \(\lambda\pi_e\) calculus, which we call the \(\lambda\pi_{e}\beta\) calculus, and show how a notion of interface may be defined for it. We show also a translation from the \(\lambda\pi_e\) calculus to the \(\lambda\pi_{e}\beta\) calculus that preserves access right requirements in a sense which we will make precise.

This chapter is organised as follows. In Section 4.1 we present the subject of resource access sharing. In Section 4.2 the notion of channel polarity is introduced, and a series of results is given concerning polarised blocking. In Section 4.3 the filter operator is introduced. Section 4.4 is dedicated to the presentation of the subject of access rights and protection domains; the related notions of capability and name space are presented and discussed. Section 4.5 is dedicated to the subject of type systems in mobile systems. In Section 4.6 we present and discuss the \(\lambda\pi_e\) calculus. In Section 4.7 a new notation is presented, the \(\lambda\pi_{e}\beta\) calculus, similar to the \(\lambda\pi_e\) calculus but where type restrictions are enforced by the filter operator. Finally, a translation from the \(\lambda\pi_e\) calculus to the \(\lambda\pi_{e}\beta\) calculus that preserves security requirements is given in Section 4.8.

### 4.1 Resource Access Sharing

We shall illustrate the problem by considering the case of resource access sharing (as presented in Milner [111]). Commonly, each process sharing a resource is assumed to behave in a way that will not disrupt the expected protocol, for example by masquerading as a service provider in order to snatch service requests sent by other agents in the system.

A resource may be represented as an agent of shape \(!x(w).R\). This agent is ready to receive along channel \(x\) a channel \(y\) instantiating \(w\) whereupon the process \(R\{y/w\}\), the resource \(R\) with its formal parameter \(w\) instantiated by \(y\), is executed. We represent a system in which a certain resource is a private resource of an agent \(P\) as the agent

\[(x)(P | !x(w).R)\]

In this system, only \(P\) is able to access the resource \(R\), although \(R\) may be able to communicate along other channels with agents in the environment of the system. However, constraints must be imposed on the behaviour of both \(P\) and \(R\) to preserve resource privacy and availability. One such constraint is that neither agent will ever extrude the restricted channel \(x\). If this would happen, \(P\) could no longer be sure of being the sole owner of \(R\). Worse yet, a mischievous agent in the environment could impersonate the resource and steal attempts by \(P\) to acquire a replica of \(R\). Another condition is that \(R\) do not bear the name \(x\), that is, that \(x\) does not occur free in \(R\) as a positive subject. The reason for this is that otherwise it might interfere with a request by \(P\) for a new replica of \(R\).

More interesting is the case in which two different agents, \(P_1\) and \(P_2\), share the private resource, as in the following system:

\[(x)(P_1 | P_2 | !x(w).R)\]

A sensible requirement we may impose on such systems is that a shared resource can be seen a private resource by any of the agents sharing it. Thus, resource sharing should be transparent for \(P_1\) and \(P_2\) in the example above, i.e. each one may regard
$R$ as a private resource. This may be formalised by requiring that the following relation holds:

$$(x)(P_1 | P_2 | !x(w).R) \sim (x)(P_1 | !x(w).R) | (x)(P_2 | !x(w).R)$$

In order for this to be true, we must impose the same restrictions on $P_1$ and $P_2$ as on $P$ above. Thus, neither process may bear the name $x$. The same holds for $R$. Also, $x$ may not be extruded by any agent in the system, and thus may not occur free as the object of an output action. All this amounts to saying that $x$ may be used by $P_1$, $P_2$ and $R$ only to access the resource, i.e. only as the subject of output actions.

A generalisation of the requirement that a private resource can be distributed over composition is to demand that it also distributes over replication, i.e.

$$(x)!P | !x(w).R) \sim !(x)(P | !x(w).R))$$

For this relation to hold, we must also require that $x$ only occurs free in either $P$ and $R$ as the subject of an output action.

These properties have very wide applications. For instance, in the reduction from the higher-order $\pi$-calculus with blocking to the first-order $\pi$-calculus in Section 3.3.3, the higher-order transmission of an agent $P$ is encoded as a replicated resource offering the first-order transmission of an encoding of $P$ instead. But the reduction is proved correct only according to a reduction semantics in which processes are assumed to be well-sorted, and in a context in which only certain agents are allowed, i.e. agents that can be obtained by reduction from the higher-order $\pi$-calculus.

A similar limitation appeared in Thomsen’s translation of Plain CHOCS to mobile processes. In this translation, whenever a process $P$ is sent in Plain CHOCS, a link to a trigger construct providing copies of $P$ is sent in the translation. The translation is a homomorphism for all operators except higher-order output prefix, which slightly modified can be defined as follows:

$$[[\pi P.P']] = (x)(\pi x.([[P']]) | !x([P']))$$

where $x$ is assumed to be fresh and $[[P']]$ denotes the translation of $P$.

Replication is needed here because the recipient may execute $P$ several times. Thomsen fails to prove that $P \approx Q \iff [[P]] \approx [[Q]]$

where $\approx$ is defined as observation congruence. However, he conjectures that this could hold under certain restrictions on the observations allowed.

By defining a new version of strong bisimulation, Sangiorgi [149] proved this equivalence for processes of arbitrary high order that respect a particular finite-order sorting. Milner conjectures [111] that this restriction does not impair the generality of the result, arguing that useful processes always respect some sorting, and also that a process that respects some kind of sorting would also respect a finite sorting. The question is whether these assumptions are valid in the context of open systems.

This kind of problem appeared already in Milner’s encodings [110] of the lazy $\lambda$-calculus and the call-by-value $\lambda$-calculus. We describe here briefly one of the versions of the encoding of the call-by-value $\lambda$-calculus.

Let $[[M]]$ denote the encoding of the term $M$ in the $\lambda$-calculus into a term in the $\pi$-calculus. The encoding is defined as follows:

\begin{align*}
[[\lambda x.M']] &= \pi x.([[M']]) &[[\lambda x.M']_y] &= [[M']_y] \\
[[M_1 \cdot M_2]] &= [[M_1]] \\
[[!M]] &= \pi x.([[M]] | !x([M]))
\end{align*}

where $x$ is assumed to be fresh and $[[M']]$ denotes the translation of $M$.
The name \( p \) in \( [[M]](p) \) is used for the transmission to the "arbiter process" \( \text{App} \), of a trigger to the \textit{value-body} of \( [[M]] \), and at the same time for signaling that \( M \) has been reduced to a value, an abstraction. The process \( \text{App} \) forces the correct interaction between application and argument.

Milner left open the question of the correctness of this encoding. Later, Sangiorgi [149] showed that in the standard \( \pi \)-calculus \( \beta \)-reduction is not valid for this encoding, although it yields a precise operational correspondence. The problem is that an agent within the external environment might interfere with the protocol by obtaining a trigger and then using it as the subject of an input action. Basically, the counter-example presented by Sangiorgi can be reduced to the following. Let

\[
A = (y)!((xy.y(w)).R), \quad B = (!y)(xy.y(w)).R).
\]

Agent \( A \) continuously transmits a trigger \( y \) to process \( R \), whereas \( B \) continuously transmits \textit{fresh} triggers to \( R \). In the standard \( \pi \)-calculus these agents are not observationally equivalent, for obvious reasons. Nevertheless, in the absence of the matching operator, if the context is not allowed to use the trigger as the subject of input actions, then both agents are indeed undistinguishable.

In order to enforce such a discipline, Sangiorgi and Pierce [134] proposed an extension of Milner’s type system in which the channels might be restricted to be used only in output actions. As a result, a heavy side condition could be avoided on the use of names in \( P \), \( Q \) and \( R \) in expressions of type \( (x)(P \mid Q \mid !x(w).R) \). The condition was the requirement that \( x \) does not occur as the subject of an output action in this expression, thus guaranteeing distributivity of composition. With this restriction, the encoding of the \( \lambda \)-calculus shown above was proved correct.

These ideas were further developed by Sangiorgi [150]. A particular set of names is declared as \textit{uniform} receptive, and a name \( x \) is considered receptive in an agent \( P \), if \( P \) is always ready to accept an input at \( x \), and all these inputs have the same continuation. Receptiveness is guaranteed by the requirement that a name be immediately available in input-replicated form as soon as it is created, while uniformity is guaranteed by demanding that the name occur only once in input position. In order to preserve uniformity, it is required also that only the output capability of the name can be transmitted.

The purpose of that work was to develop proof techniques to solve some of the problems mentioned above: proofs of correctness of transformations requiring distributivity of composition in resource sharing; proofs of correctness of higher-order reductions of the \( \pi \)-calculus to first-order; and proofs of correctness of Milner’s encoding of the call-by-value \( \lambda \)-calculus into the \( \pi \)-calculus. These problems derive from the fact that there are contexts in the standard \( \pi \)-calculus which are able to distinguish between processes which we want to regard as equivalent. This is not possible if we only allow “well-behaved” agents to take part in these contexts. By imposing a type system, some kinds of undesirable contexts may be ruled out as ill-typed.

In the join calculus [47], all names can be regarded as uniform receptive in some sense. A join calculus definition of type
might be encoded in the asynchronous $\pi$-calculus as
\[(xy)!((x(u), y(v), uv) | \tau(xy))\]

This agent extrudes two fresh channels $x$ and $y$ along $z$, and then it is ready to continuously receive two channels $u$ and $v$ along $x$ and $y$, whereupon it offers $v$ along $u$, $uv$.

Although in this translation the symmetry between $x$ and $y$ is lost, this fact cannot be observed in an asynchronous setting. However, this translation is not correct if the agent occurs within a context in which another agent may start reading values along either $x$ or $y$. The problem disappears if we adopt a type system with polarities, since then we could let $z$ transmit only the output capabilities of $x$ and $y$. An alternative to this solution was presented by Fournet and Gonthier [47]. Instead of introducing typing rules a firewall is built around the process, consisting of one-way relays for every free name in the agent, and for every name that crosses the firewall. Only proxies to channels are forwarded to the context or received from the context. In this way, only fresh names are transmitted, and using them in input position would be no different than using any other fresh name instead. The semantics of the names transmitted are thus wholly determined by the sending agent, which is the only agent that knows about the dependencies among these names.

In both methods presented here, i.e. enforcing a type discipline resp. setting up firewalls, the flexibility in the use of names, a crucial feature in open systems, is impaired. Tagging names with polarities impose restrictions on the use of names which is hard or maybe even impossible to enforce in open systems; setting up firewalls of the kind described here might isolate an agent excessively.

### 4.2 Polarised Blocking

It is not our intention to produce new proofs of correctness of the encodings described above. We want only to be able to disallow certain operations in order to preserve the semantics of resource access. We can do this by introducing primitive operators that are not more restrictive than necessary. We should for instance be able to guarantee immediate resource access without thereby restricting the freedom and flexibility in the use of names by the system.

What is called for are operators that restrict the communication capabilities of agents, as the blocking operator, but that block synchronisation along a channel only in one direction, either input or output. We may call these operators positive blocking resp. negative blocking. The blocking operator may thus be expressed as a combination of such operators disallowing synchronisation in both directions.

We extend thus the syntax of the standard $\pi$-calculus with two constructs: for any agent $P$, let $P\setminus x^+$ denote positive blocking on channel $x$, and $P\setminus x^-$ negative blocking. Positive blocking on $x$ means that any attempts by observers to do an input along $x$ with an agent within the scope of blocking will fail. Negative blocking means the same for output attempts. The transition rules $\text{BLOCK}^+$ and $\text{BLOCK}^-$, may thus be defined as follows:

\[
\text{BLOCK}^+ : \quad \frac{P \xrightarrow{\mu} P'}{P\setminus z^+ \xrightarrow{\mu} P'\setminus z^+}, \quad \mu \text{ not an output action along } z
\]
The algebraic rules associated with these operators are similar to the blocking operator, which can now be defined as follows:

\[ P \setminus x := P \setminus x^+ \setminus x^- \]

We must now show how these operators may be used in order to obtain distribution of a private resource over composition, without at same time imposing further conditions on the use of triggers to the resources. We must permit the extrusion of triggers, thus allowing external agents to access those resources. At the same time, we must guarantee resource access by internal agents. A preliminary solution would be a system that disallows input actions along the resource trigger from the “outside”:

\[ (x)((P \vert Q \vert !x(w).R)\setminus x^+) \]

Since the trigger \( x \) may be extruded in any case, we may simplify the system to

\[ (P \vert Q \vert !x(w).R)\setminus x^+ \]

An external agent may thus access the resource \( R \) along \( x \) in this system, but it will not be able to interfere with resource access requests by either \( P \) or \( Q \). But one problem remains: either \( P \) or \( Q \) might disrupt the protocol if any of them offers an input along \( x \), thus interfering with an eventual resource request by other agents. The same applies to agent \( R \) or any of its derivatives. It is not enough to demand that \( x \) does not occur syntactically in any of these agents, since names bound by the input prefix, which could eventually be instantiated to \( x \), may occur. We must therefore prevent these agents from synchronising with each other along \( x \). This may be achieved by blocking any input attempts by these agents along \( x \). Thus, the system we want is

\[ (P \setminus x^- \vert Q \setminus x^- \vert !x(w).R\setminus x^-)\setminus x^+ \]

To simplify notation we introduce the following definitions:

\[ \hat{P} := (P \setminus x^- \vert !x(w).R\setminus x^-)\setminus x^+ \]

\[ \hat{P}_1, \ldots, \hat{P}_n := (P_1 \setminus x^- \vert \ldots \vert P_n \setminus x^- \vert !x(w).R\setminus x^-)\setminus x^+ \]

\[ \hat{!P} := (!P \setminus x^- \vert !x(w).R\setminus x^-)\setminus x^+ \]

for any agents \( P, P_1, \ldots, P_n \).

We define also

\[ \bigwedge_{i \in I} \hat{P}_i := \hat{P}_{a_1}, \ldots, \hat{P}_{a_m} \]

for all \( a_j \in I \) and \( I \) a finite index set.

The result we want to obtain may thus be expressed as
Likewise, we want to prove that
\[
\widehat{\hat{P} \mid \hat{Q}} \sim \hat{P} \mid \widehat{\hat{Q}}
\]
Proofs of these assertions will be given below.

These operators can be expressed in the standard \( \pi \)-calculus with matching and mismatching. The encoding is similar to that of the blocking operator, but a little more complex. We have to distinguish the treatment of the input prefix and the output prefix.

The encoding of positive blocking is as follows. Let \( P \) be any process; we define a transformation \( T^+ \) on agents that is a homomorphism over all operators except positive blocking, in which case it is defined as follows:

\[
T^+\left(P \mid z^+\right) = (w)T^+_{wz}(P), \ w \text{ fresh}
\]

Intuitively, \( T^+_{wz}(P) \) performs the task of testing the subject of an input action prefix \( \alpha \), in an agent \( \alpha P \), for equality with \( z \), and then creates a new process consisting of a choice between two copies of the translation of \( P \), one prefixed by \( \alpha \) and the other by \( \alpha \{w/z\} \). The former keeps the resource available for agents in the environment, and the latter to processes occurring within the scope of the blocking operator. For output prefixes, all occurrences of \( z \) as the subject of the prefix are simply substituted by \( w \), thus preventing outsiders to interfere with the resource access request. Assuming processes are conveniently renamed to avoid name clash between \( w, z \) and any other bound channel, \( T^+_{wz} \) is defined as a homomorphism over all operators except positive blocking and prefix:

\[
T^+_{wz}(xy.P) = [x = z]xy.T^+_{wz}(P) + [x \neq z]xy.T^+_{wz}(P)
\]
\[
T^+_{wz}(x(y).P) = x(y).T^+_{wz}(P) + [x = z]w(y).T^+_{wz}(P)
\]
\[
T^+_{wz}(P \mid x^+) = T^+_{wz}(T^+(P \mid x^+))
\]

For negative blocking the transformation is similar, except that the treatment for input and output prefixes is reversed. Thus, we define \( T^- \) as homomorphism over all operators except negative blocking, in which case it is defined as follows:

\[
T^-\left(P \mid z^-\right) = (w)T^-_{wz}(P), \ w \text{ fresh}
\]

Assuming processes are conveniently renamed to avoid name clashes between \( w, z \) and any other bound channel, \( T^-_{wz} \) is defined as a homomorphism over all operators except negative blocking and input prefix:

\[
T^-_{wz}(xy.P) = xy.T^-_{wz}(P) + [x = z]xy.T^-_{wz}(P)
\]
\[
T^-_{wz}(x(y).P) = [x = z]w(y).T^-_{wz}(P) + [x \neq z]x(y).T^-_{wz}(P)
\]
\[
T^-_{wz}(P \mid x^-) = T^-_{wz}(T^-(P \mid x^-))
\]

We now prove the two propositions mentioned earlier.

**Proposition 1:**
\( \hat{P}, \hat{Q} \sim \hat{P} \mid \hat{Q} \)

**Proof:** Let \( \{P_1, \ldots, P_n\} \) be any finite set of agents, \( S = \{1, \ldots, n\} \), and let \( S_1 \) and \( S_2 \) be a partition of \( S \) into two disjoint sets.

We define a relation \( B \) consisting of all pairs of type

\[
\left( \bigwedge_{i \in S} P_i, \left( \bigwedge_{i \in S_1} P_i \bigm\| \bigwedge_{i \in S_2} P_i \right) \right)
\]

for any vector name \( \vec{y} \), any \( n \in N \) and any set of agents indexed by \( \{1, 2, \ldots, n\} \).

We may assume that all channels bound by a restriction operator have been moved outside and occur in \( \vec{y} \), after convenient renaming. We will show that \( B \) is a strong bisimulation. The proof of the proposition will then have been established, since

\[
\left( \hat{P}, \hat{Q}, \hat{P} \mid \hat{Q} \right) \in B.
\]

Let \( (P, P') \in B \). Then, for some \( n \in N \) and name vector \( \vec{y} \), we must have

\[
P = \bigwedge_{1 \leq i \leq n} P_i, \quad P' = \left( \bigwedge_{i \in S_1} P_i \bigm\| \bigwedge_{i \in S_2} P_i \right)
\]

where \( S_1, S_2 \) is a partition of \( S = \{1, 2, \ldots, n\} \).

Assume \( P \xrightarrow{\mu} Q \). There are several alternatives:

(i) \( \mu \) is a free output along a channel \( z \neq x \). Then we must have, for some \( j \in S \),

\[
P_j \xrightarrow{\mu} P'_j, \quad Q = \left( \bigwedge_{i \in S_1} P'_i \right)
\]

where \( P'_i = P_i \) except for \( i = j \). Either \( j \in S_1 \) or \( j \in S_2 \). Assume without loss of generality the former. We obtain

\[
\bigwedge_{i \in S_1} P_i \xrightarrow{\mu} \bigwedge_{i \in S_1} P'_i
\]

Hence,

\[
P' = \left( \bigwedge_{i \in S_1} P_i \bigm\| \bigwedge_{i \in S_2} P_i \right) \xrightarrow{\mu} Q' \equiv \left( \bigwedge_{i \in S_1} P'_i \bigm\| \bigwedge_{i \in S_2} P'_i \right)
\]

It is now easy to see that

\( (Q, Q') \in B \)

(ii) \( \mu \) is an input along a channel \( z \neq x \). The argument is similar, since \( B \) is closed under substitutions.

(iii) \( \mu \) is a bound output along a channel \( z \neq x \). The argument is similar to (i). The object of the action, say \( v \), must be a name in \( \vec{y} \), and \( Q \) must be an agent restricted
by \( \vec{y}' \), a name vector equal to \( \vec{y} \) with \( e \) deleted. It is easy to verify, as we did in case (i), that \( P' \) may execute the same transition, yielding a matching agent.

(iv) \( \mu \) is an output along \( x \): this is impossible.

(v) \( \mu \) is an input along \( x \), for instance \( x(v) \). Then

\[
P = (\vec{y}) \bigwedge_{1 \leq i \leq n} P_i \xrightarrow{\mu} Q \equiv (\vec{y}) \left( \bigwedge_{1 \leq i \leq n+1} P_i \right)
\]

where \( P_{n+1} = R\{v/w\} \). Now, either \( \bigwedge_{i \in S_1} P_i \) or \( \bigwedge_{i \in S_2} P_i \) may execute the same transition. Assume the former, without loss of generality. Then, we must have

\[
\bigwedge_{i \in S_1} P_i \xrightarrow{\mu} \bigwedge_{i \in S'_1} P_i
\]

where \( S'_1 = S \cup \{n + 1\} \). Also

\[
P' \xrightarrow{\mu} Q' \equiv (\vec{y}) \left( \bigwedge_{i \in S'_1} P_i \big| \bigwedge_{i \in S_2} P_i \right)
\]

We can now easily check that \( (Q, Q') \in B \).

(vi) \( \mu \) is the silent action, involving only one agent \( P_k \). In this case the same agent in \( P \) may execute a silent transition yielding a matching pair.

(vii) \( \mu \) is the silent action, involving two agents \( P_k \) and \( P_m \). Then, the communication must happen along a channel \( z \neq x \). In this case, \( P_k \) and \( P_m \) may also synchronise along the same channel in \( P' \), and we can check that the result is a matching pair.

(viii) \( \mu \) is the silent action, involving an agent \( P_k \) and \( !x(w).R \). Then, if the output by \( P \) is \( \overline{\alpha}v \), we get

\[
P_k \xrightarrow{\overline{\alpha}v} P'_k, \quad P \xrightarrow{\tau} Q \equiv (\vec{y}) \bigwedge_{1 \leq i \leq n+1} P'_i
\]

where \( P'_i = P_i \) for \( i \neq k, n + 1 \), \( P'_{n+1} = R\{v/w\} \). Now, either \( i \in S_1 \) or \( i \in S_2 \). Assume the former, without loss of generality. Then

\[
\bigwedge_{i \in S_1} P_i \xrightarrow{\tau} \bigwedge_{i \in S'_1} P_i
\]

where \( S'_1 = S \cup \{n + 1\} \). Hence,

\[
P' \xrightarrow{\tau} Q' \equiv (\vec{y}) \left( \bigwedge_{i \in S'_1} P_i \big| \bigwedge_{i \in S_2} P_i \right)
\]

and \( (Q, Q') \in B \) as desired.

The proof in the other direction is not harder, and the argument in each case is similar.

**Proposition 2:**

\[
\hat{\overline{!P}} \sim !\hat{P}
\]
Proof: We define a relation \( \mathcal{R} \) consisting of all pairs of form

\[
\left( (\bar{y})(P/x^-) | P_1/x^- | \ldots | P_n/x^- | x(w).R/x^-) \right) x^+,
\quad (\bar{y}) \left( P \big| \bigwedge_{i \in S_1} P_i \big| \ldots \big| \bigwedge_{i \in S_m} P_i \right)
\]

for any agents \( P_1, \ldots, P_n \), any \( n \in \mathcal{N} \), and any partition \( S_1, \ldots, S_m \) of the set \( \{0,1,\ldots,n\} \). By taking \( n = 0 \), we see that \( (\bar{P}, \widehat{P}) \in \mathcal{R} \). We get the result if we can show that \( R \) is a bisimulation. The proof, which we omit here, is a case by case study of all possible transitions, similar to the proof of Proposition 1.

### 4.3 The Filter Operator

An alternative to the blocking operator is the filter operator, which will be needed in Sections 4.7 and 4.8. We write \( P[L] \) to denote the process \( P \) filtered by \( L \), a finite set of polarised channel names. The meaning of this operator is given by the rule:

\[
\text{FILT: \quad } P \xrightarrow{\mu} P' \quad \quad P[L] \xrightarrow{\mu} P'[L]
\]

\( \mu \) an output action along a channel \( z \) s.t. \( z^+ \in L \), or

\( \mu \) an input action along a channel \( z \) s.t. \( z^- \in L \)

The process \( P[L] \) may be viewed as the process \( P \) blocked by all polarised names that do not occur in \( L \).

Many algebraic properties associated with the blocking operator are preserved by the filter operator. Thus, if \( P \simeq Q \) then \( P[L] \simeq Q[L] \). The following equivalences also hold:

\[
\begin{align*}
(a) \quad (x(y).P)[L] \simeq x(y).(P[L]) & \quad \text{if } x^- \in L \text{ and } \{y^+,y^-\} \cap L = \emptyset \\
(a') \quad (\exists y.P)[L] \simeq \exists y.(P[L]) & \quad \text{if } x^+ \in L \\
(b) \quad (x(y).P)[L] \simeq 0 & \quad \text{if } x^- \notin L \\
(b') \quad (\exists y.P)[L] \simeq 0 & \quad \text{if } x^+ \notin L \\
(c) \quad (P + Q)[L] \simeq P[L] + Q[L] & \\
(d) \quad P[L][L'] \simeq P[L'][L] & \\
(e) \quad (y)(P)[L] \simeq (y)(P[L]) & \quad \text{if } \{y^+,y^-\} \cap L = \emptyset \\
(f) \quad ([x = y]P)[L] \simeq [x = y](P[L]) & \\
(g) \quad P[L][L'] \simeq P[L \cap L'] & 
\end{align*}
\]

Note that (d) follows from (g).

Nevertheless, the results obtained in Section 3.2.4 showing that blocking may be expressed in terms of matching and mismatching need not be valid for filter, and the technique for showing this equivalence does not work with the filter operator. Whether the filter operator may be expressed in terms of matching and mismatching is an open question which we do not investigate here.

### 4.4 Access Rights and Capabilities

Sharing of resources is one of the most important motivating factors in distributed systems. The architecture of distributed systems is usually described in terms of processes that encapsulate objects or resources, allowing clients to access these resources along interfaces via interactions [37].

60
In this architectural model, security is enforced by protecting the objects encapsulated by a process against unauthorised access, as well as by securing the channels used for interactions among processes. Authorised access is supported by the use of access rights specifying for each process which operations on objects it is allowed to invoke.

Recently, programming languages have been designed to support mobile code, i.e. higher-order code that is transferred from a remote location or domain and executed within the local environment. This may expose the internal interfaces and objects within a location to attacks by mobile code. There are several techniques for securing mobile code. These techniques are usually based on the administration of capabilities or privileges assigned to processes, and are enforced either statically (during compilation), or dynamically (at run time).

A technique for statically checking mobile code is based on the notion of typing and type safety. However, static checking of properties is not sufficient to ensure safety. In Java, for instance, a scheme called the Sandbox model of protection against mobile code was developed. This model includes both static and dynamic checking. The Java Virtual Machine (JVM) gives each application its own runtime environment and a security manager that restricts the resources available to the process.

We should distinguish here between typing and capability, and analogously between typing errors and access rights violations. The notion of type in programming languages usually refers to data types, and type safety to memory safety, control flow safety, or data type safety, e.g. integrity of data-abstraction boundaries. By contrast, access right violations are not typing errors in the strict sense of the word.

Typing errors may be avoided by static type checking or by dynamic runtime checking. However, to our knowledge, in current systems access rights are always enforced dynamically, e.g. by runtime monitoring of process execution. The reason seems to be that access rights and capabilities are basically dynamic features. A process may extend its behaviour in several ways, e.g. by invocation of procedures from a library, or by communication that may alter the information available to a process. This makes it very hard to decide at compile time whether some disallowed operation will be undertaken at runtime. Access control by static means have nevertheless been the subject of recent research, e.g. certified code [120, 122]. However, current certifying compilers have so far concentrated on standard type and memory safety [121]. Other approaches involving certified code concentrate exclusively on standard type safety [87, 117]. Only very recently there have been attempts at formulating a type system for enforcing access control, e.g. security automata [181]. Nevertheless, runtime checks are not eliminated thereby. For instance, checks may be placed by the compiler after a local file access if the security policy states that the network cannot be be used when local files have been read. Another recent work along these lines is Skalka [156]. It is perhaps the first work where static type systems are used to enforce fine-grained access control. It introduces the notion of security type system for enforcing safety “with respect to certain access control properties at runtime.” An attempt to encode access control in a type system by extending it with dependency information is Potter [138].

An example of runtime access control is the Java language’s access semantics, which is enforced by the bytecode runtime system, that throws an exception if a rogue applet attempts to call a private method. Another example is the security manager introduced in the beta release of JDK, which may be regarded as a reference monitor, as defined by Lampson in his classical paper on protection domains [91]. The security manager implements a security policy and centralises all access control
decisions. This security architecture has been adopted by Netscape. The security manager throws an exception for all access checks. The type system of Java is used basically to provide protection for the security manager, and is thus only an aid to the manager. The idea is that if the type system is correct, the security manager will be safe. The access control mechanism must nevertheless be ultimately performed by the security manager at runtime.

Although type systems for access control is still an incipient area of research, with no practical technology yet available, in process algebra calculi access rights have been encoded exclusively in terms of types, either channel types denoting capabilities ([134, 64, 126]), or process types denoting the interface of a process [187]. It is claimed that typing can be used to control the effect of mobile code on local environments. This is certainly true in relation to the integrity of certain kind of data. However, as we saw above, it is uncertain whether this also applies to access control. Nevertheless, it is almost exclusively in terms of these notions that type systems have been developed in process algebraic calculi. By contrast, in programming languages protection mechanisms of this kind have been enforced exclusively, as far as we know, in terms of runtime control and monitoring, cf. Java’s sandbox model. In spite of this, no attempts seem to have been made to encode this kind of mechanisms in a process algebra notation.

This discrepancy might be due to limitations in the expressiveness of current process algebra notations, specially in relation to the notion of context or domain that restricts the capabilities and privileges of a process on resource consumption. Moreover, the typing approach also has many limitations. The very notion of capability associated to channels or processes, as it is currently defined, seems to be questionable. We return to this point below. What seems certain is that access control safety can at best be enforced by a mix of static and dynamic techniques, which implies that the latter cannot be abolished altogether in practice.

### 4.4.1 Protection Domains

A capability is conceptually a token (name, key, password, ticket) that gives permission to its possessor to access an entity, object or resource in a computer system. Capabilities are implemented as data structures containing at least two items: a unique object identifier naming or addressing a single object in a computer system, and the access rights associated with this object. An object can be any logical or physical entity, and the access rights define the operations that can be performed on it. A process in a capability system may be assigned a list of capabilities specifying the operations that the process is allowed to perform on the objects to which it has access. An alternative is to associate an access list to each object in the system specifying which processes or users may access the object.

Capabilities are the basis for object protection. A process may access an object or perform an operation upon the object only if it is assigned the capability or privilege for that object and operation. Capability and access lists must be administered by the system, and may not be modified by the process itself.

The conceptual basis for protection and access control was laid by Lampson [91], who introduced the notion of protection domain. Lampson mentions as related notions those of environment, state, sphere, capability list, ring and domain. A protection domain is an execution environment shared by a number of processes. It may consist of a capability list specifying for each resource the kind of operations that the processes executing within the domain are allowed to perform on the resources, i.e. the access rights associated with the domain. A capability list is associated with each user in the system.
This scheme contrasts with the approach of enforcing protection by maintaining *access control lists* associated with each object in the system, not with each user. In this scheme, as mentioned above, there is a list associated with each object or resource in the system, containing the names of users and their privileges concerning the operations on the object.

In this context it is important to emphasise that a capability commonly implies the possession of some token (a key, a password, a ticket) giving automatic access rights to whoever possesses it.

A protection system may be described as a system consisting of a set of objects, a set of domains, and an access matrix. Each domain may be endowed with a different access policy specified in the corresponding entry in the access matrix. For each access of an object, the system must check that the protection policy associated with the access rights of the protection domain is not violated.

Protection domains are outlined within the framework of an idealised system called the *message system*, which consists of processes that communicate with each other only via message passing. Each process represents a single domain, isolated from other processes except for message transmission. According to Lampson, such a process may be viewed as a separate machine consisting of a set of resources such as memory, file storage, tape units, etc.

Although originally conceived mainly for modular programming, the notion of protection domain is perhaps even more relevant in the context of code mobility. It provides the basis of protection for Java programs that include mobile code. Like local code, downloaded code is provided with a protection domain specifying its access rights. Thus, if we admit that code may be communicated in a message, and thereafter executed, we need a notion of protection that is based not on the protection offered by message passing, but on the notion of access control to resources which may be invoked by direct access, as is the case in conventional operating systems. In other words, we want to restrict access to resources by preventing access right violations by monitoring the executing process.

We may thus assume that resources are *directly* accessible, i.e. by invocation of operations associated with the resource. A resource may be viewed as a process that is unprotected and directly accessible.

A protection domain is an abstract concept which may be implemented in many ways. We propose here to encode the notion of protection domain in a π-calculus notation extended with operators for polarised blocking, and in a framework that includes code mobility. We illustrate this by a simple example which we give in the next section. The technique of blocking forbidden operations according to the capabilities assigned to the executing process may be compared to the implementation of access restriction by process-local segment tables, which consists of a list of descriptors defining the memory segments available to a program. A process may access segments by virtual addressing. A descriptor contains the physical location of the segment, its length, and a field specifying the rights or type of access permitted to that segment. The hardware ensures that only allowed operations are executed, otherwise an error is signaled. This scheme is closely related to the notion of blocking certain actions. The program is executed in a static addressing environment.

### 4.4.2 Name-Space Management

A security policy can be also enforced by an approach to security architecture which is closely related to traditional capability systems, but which is based on the notion
of name space management. This approach highlights several aspects related to the notion of name spaces that are central to this thesis. Basically, a name space contextualises the meaning of names. The blocking operator can be said to have a dual nature. On the one hand, it simply blocks external communication along certain channels. But not less important is the apparently trivial fact that it allows internal communication along blocked channels. This second aspect is important because, by allowing communication along blocked channels, the blocking operator is in fact contextualising the meaning of blocked names. As a result, it may be used to encode name spaces, and hence also dynamic binding and dynamic linking of names, which are essential to describe many features of distributed systems. This is also the gist of the name space approach to security.

As an example we can mention an important property of Java's dynamic linking, namely the ability to create environments where different applets see distinct classes with the same names [183]. This ability is used to limit the capabilities of a Java applet by restricting its name space. A security policy is thus enforced in Java by controlling how names are resolved into runtime classes, causing them to refer to any desired class. The technique is also used in Safe-Tel [25] to hide commands in an untrusted interpreter, and in Plan 9 [136] to attach different programs and services to the file system seen by an untrusted agent.

Classes may be seen as resources in object-oriented systems. Access to a file system can be restricted by making the name of its file class invisible to the remote code linked to local classes. This would make attempts to access the file to fail or raise an exception. More interestingly, the remote code may be executed in an environment where sensitive classes have been replaced with compatible ones. The effect is tantamount to giving different capabilities to system resources, with the difference that names space is static and thus cannot be passed via communication, in contrast to capabilities. Security decisions are thus made statically, before execution of the remote code or applet.

An environment that maps class names to their implementations is referred by Wallach [183] as a configuration. Every class can have a different name space configuration, and in this way "we can arrange for the new classes to see the original sensitive classes, and we can block applet classes from seeing the same sensitive class" [183].

Name space management increases performance and in this way avoids one of the main problems related to runtime security checks. Hence, it is not always true that blocking, as defined here, in contrast to typing encodes dynamic, runtime checks. Blocking may also be used to encode the notion of name space, which is at the root of dynamic binding in object-oriented languages. We will see many examples related to this in Chapter 5.

4.4.3 Protection Domains: An Example

We illustrate the protection model with a simple example, consisting of a site or principal that accepts mobile code or scripts. Upon receipt of a request to execute mobile code, the principal chooses an appropriate security policy according to the group identity of the sender. We assume that the principal trusts the group identity specified in the request message, which can be seen as a kind of key which has been distributed in a way that need not concern us here. The security policy is implemented as a blocking policy, where access to certain resources (memory, processes, libraries) is restricted according to the access rights assigned to the group.

The principal consists of two registers, Reg1 and Reg2, a library Library, and a
process \texttt{Process} that can be awakened on demand. For the sake of simplicity, we let registers, instead of files, represent memory. A register is simply a process that accepts read and write requests. The definition of register \texttt{Reg}_1 is the following:

\[
\texttt{Reg}_1 \overset{\text{def}}{=} (\texttt{val}) \\
\left(\texttt{write}_1(x) \cdot \texttt{val}(y) \cdot \texttt{val}(x) \cdot 0 \mid \\
\texttt{read}_1(r) \cdot \texttt{val}(y) \cdot (r(y) \cdot 0 \mid \texttt{val}(y) \cdot 0 \mid \\
\texttt{val}(0) \cdot 0 \mid \\
\texttt{val}, \texttt{write}_1, \texttt{read}_1)\right)
\]

The value \(v\) stored in the register is represented by the process \(\texttt{val}(v) \cdot 0\), where \(\texttt{val}\) is a fresh name with only local significance. The register may be accessed via the channels \texttt{write}_1 for writing and \texttt{read}_1 for reading. Only outputs to these channels are accepted. The definition of register \texttt{Reg}_2 is analogous. Both registers are initialised to the value 0.

The definition of the process \texttt{Process} is as follows:

\[
\texttt{Process} \overset{\text{def}}{=} (\texttt{ready}, \texttt{execute}, \texttt{reply}) \\
\left(\texttt{ready}, \texttt{proc}(a, r) \cdot \texttt{execute}(a) \cdot \texttt{reply}(\texttt{res}) \cdot \texttt{ready} \cdot \texttt{res} \cdot 0 \mid \\
\texttt{execute}(a) \cdot \texttt{reply}(\texttt{res}) \cdot 0 \mid \\
\texttt{ready} \cdot 0 \mid \\
\texttt{ready}, \texttt{execute}, \texttt{reply}, \texttt{proc}^+\right)
\]

The dots in the expression \(!\texttt{execute}(a)\ldots\texttt{reply}(\texttt{res})\cdot 0\) denote the behaviour of the process, which is left unspecified. The interface of the process is the channel \texttt{proc}, which has two parameters, \(r\) for the address where the result should be sent, and \(a\) for the argument value. Upon completion of an invocation the result is sent, and a message \texttt{ready} signals completion, whereupon the process is ready to accept new invocations.

The library \texttt{Library} consists of two procedures or methods, \(m_1\) and \(m_2\), whose code is represented by \texttt{code}_{m_1} resp. \texttt{code}_{m_2}.

\[
\texttt{Library} \overset{\text{def}}{=} \\
\left(\!\texttt{lib}(m, r) \cdot \texttt{lib}(r) \cdot 0 \mid \\
\texttt{m}_1(r) \cdot \texttt{r}(\texttt{code}_{m_1}) \cdot 0 \mid \\
\texttt{m}_2(r) \cdot \texttt{r}(\texttt{code}_{m_2}) \cdot 0 \mid \\
\texttt{m}_1, \texttt{m}_2, \texttt{lib}^+\right)
\]

The \texttt{Library} accepts requests on a channel \texttt{lib}, whose parameters are the name of the requested method, \(m\), and the reply address, \(r\). The code corresponding to the requested method is sent along the reply channel.

The resulting system, \texttt{Domain}, which include the resources defined above, is specified as follows:

\[
\texttt{Domain} \overset{\text{def}}{=} \\
\left(\!\texttt{d}(\texttt{group}, \texttt{script}) \cdot \\
\texttt{group} = a_1[\texttt{script} \cdot \texttt{ap}_1] \mid \\
\texttt{group} = a_2[\texttt{script} \cdot \texttt{ap}_2] \mid \\
\texttt{group} = a_3[\texttt{script} \cdot \texttt{ap}_3] \mid \\
\texttt{Reg}_1, \texttt{Reg}_2, \texttt{Process}, \texttt{Library} \mid \texttt{intf}\right)
\]

65
where

\[
\begin{align*}
ap_1 & \overset{\text{def}}{=} \{ \text{read}_{r1}, \text{write}_{r2}, \text{read}_{r2}, \text{write}_{r1}, \text{proc}, \text{lib} \} \\
ap_2 & \overset{\text{def}}{=} \{ \text{read}_{r2}, \text{write}_{r2}, \text{read}_{r1}, \text{write}_{r1}, \text{proc}, \text{lib}^{-} \} \\
ap_3 & \overset{\text{def}}{=} \{ \text{read}_{r1}, \text{read}_{r2}, \text{write}_{r2}, \text{write}_{r1}, \text{proc}, \text{lib}^{-} \} \\
\text{intf} & \overset{\text{def}}{=} \{ d^{+}, \text{write}_{r1}, \text{write}_{r2}, \text{read}_{r1}, \text{read}_{r2}, \text{proc}, \text{lib} \}
\end{align*}
\]

As we may observe from the definition, the principal accepts requests to execute mobile code along channel \( d \), the identity of the principal. Requests include two parameters, the name of the group to which the requesting agent belongs, and the code to be executed. The principal checks the name of the group prior execution of the code or script, which takes place within the scope of a blocking operator specifying the access rights assigned to the group.

There are three different access policies, one for each group. The access policy of the first group, \( a_1 \), specifies that scripts belonging to this group may execute within a sandbox restricting the privilege to read from register \( \text{Reg}_{1} \), to write to register \( \text{Reg}_{2} \), and to invoke the process \( \text{proc} \) or any methods from the library. Objects belonging to this group may nevertheless read from register \( \text{Reg}_{2} \) and write to register \( \text{Reg}_{1} \). Objects belonging to the second group may execute the process, as well as read from and write to register \( \text{Reg}_{1} \). The third group may access the library, execute the process, and write to register \( \text{Reg}_{1} \).

A more conventional approach in process algebra is to describe such access policies in a type system. The problems of that approach are discussed in Section 4.6.7.

### 4.5 Type Systems for Mobile Processes

Milner [111] presented a type system that assigned arities to channels and enforced a typing discipline in their use. Sort or type information was assigned only to channels, whereas processes were declared only as either well-sorted or not well-sorted.

This type system was refined by Pierce and Sangiorgi [134] by the introduction of the notion of input and output capabilities for channels, as well as by replacing by-name matching of sorts with structural matching. The notion of sorts was refined to include a form of subsorting in which the use of channels in the subject position of a prefix action may be restricted in a given context to either input-only or output-only.

The subsort relation was motivated by the situation in which several processes share a resource, e.g. a printer. If printer requests are submitted along a channel \( p \) representing the printer, a client process \( C_1 \) that submits to the printer a sequence of two jobs \( j_1 \) and \( j_2 \), which are values of some sort \( T \), may be encoded as follows:

\[
C_1 = \pi(j_1) \pi(j_2) \cdot 0
\]

The complete system may be encoded as

\[
(p : (T))(P | C_1)
\]

where \( P \) denotes the printer process. In this system, the print jobs \( j_1 \) and \( j_2 \) are received and processed in that order.
However, in the presence of another process sharing the printer, this need not be the case. If we define a misbehaving agent $C_2$ by

$$C_2 = p \langle j : T \rangle . 0$$

then the executing program

$$\langle p : (T) \rangle (P \mid C_1 \mid C_2)$$

cannot guarantee that the jobs $j_1$ and $j_2$ will be processed. The agent $C_2$ may disrupt the protocol by matching printing requests by $C_1$ and then throwing them away.

To prevent this bad behaviour, three kinds of access capabilities to a channel are distinguished: the capability to read from a channel, to write to a channel, and to do both. Thus, the client processes should only be allowed to write to $p$, and the sort of $p$ in $C_2$ may be defined as $(T)^w$, meaning that it is a channel that can be used only for the output of a value of type $T$. Within the context of the printer, however, the channel $p$ is defined to be of type $(T)^{\ast}$, i.e. it can be used only for input of any value of type $T$. The type of the channel $p$ within the context of the whole system is $(T)^b$, meaning the $p$ may be used for both input and output. As the authors point out, this approach is reminiscent of the way data encapsulation restricts the visibility of local state in sequential languages [134]. The appropriate behaviour of each component of the systems can thus be statically ensured by typechecking these components in an environment where the appropriate form of access is enforced by the sort of $p$.

The subsort relation $\leq$ declares $b \leq r$ and $b \leq w$, where $r$ denotes the capability to read from a channel, $w$ the capability to write to a channel, and $b$ the capability to do both. The formal definition will not be given here. For our purposes, it suffices to say that an output operation $\exists(y)$ is well-sorted only if $S_y \leq S_x$, where $y : S_y$, $x : (S_x)^I$ and $I$ is either $b$ or $w$; likewise, an input operation $x(y : T)$ is well-sorted only if $S_x \leq T$, where $x : (S_x)^I$ and $I$ is either $b$ or $r$.

In the printer example, if we let $p$ be made known to the components at an initiation phase, the resulting system could be described as follows:

$$\langle p : (T)^b \rangle (a \langle p : (T)^r \rangle . P \mid b \langle p : (T)^w \rangle . C_1 \mid b \langle p : (T)^w \rangle . C_2 \mid \pi(p). \delta(p). \delta(p). 0)$$

Apparently, this agent is typable only if $C_2$ does not misbehave. Nevertheless, bad behaviour is in fact not prevented by this kind of typing. Suppose $C_2$ is defined as follows:

$$C_2 \overset{\text{def}}{=} b \langle p : (T)^w \rangle . c(x : (T)^b) . x(j : T) . 0$$

In this case, the system is typable, and $C_2$ could be able to receive along $c$ the channel $p$ and thus disrupt the protocol. We must thus make sure that the channel $p$ is never extruded. The problem is the rule for typing input prefixes, $\Gamma$-In, which is defined as follows:

$$\Gamma \vdash \Gamma(a) \leq (S_1, \ldots, S_n)^r \quad \Gamma, b_1 : S_1, \ldots, b_n : S_n \vdash P : ok \quad \Gamma \vdash a(b_1 : S_1, \ldots, b_n : S_n). P : ok$$

This rule offers no way of identifying the channel $b_1$ with any channel $b$ in the domain of $\Gamma$, which would be necessary if $b_1$ is instantiated to $b$ in order to discover any inconsistencies in the typing.
The adequacy in this context of the notion of capability can be questioned. What we usually obtain by typing a process is a description of some properties of the behaviour of the process, showing for instance that a process of a certain form will not execute a determined action, rather than that it will not be able to perform this action. This implies that we need to know the whole behaviour of a process, i.e. the process itself, in order to know whether it has a given capability. Hence, instead of saying that a process lacks the ability to read from a channel, we should rather say simply that it does not read from the channel. A process of form \( x(y : (T)^w), P \) is thus well-typed if \( P \) never executes an output action along channel \( y \). This is an assertion about the actual behaviour of \( P \), not about capabilities. A capability, on the other hand, should be regarded conceptually as a real entity of some kind (a token, ticket, or entry in a list used by the operating system) giving the privilege to perform a given operation. If the process does not possess a certain capability, then it will not be able to perform the operation associated with it, independently of its actual behaviour, which usually is not known in advance. Conversely, the fact that an agent is given a certain capability need not imply that this capability will ever be used by the agent.

The discrepancy between the notion of capability and actual behaviour may be best highlighted by the following example. The agent

\[
!\mathcal{P}(a).0 \mid x(y : (T)^w) \mathcal{G}(b).0 \mid x(y : (T)^b) y(z : T) \mathcal{G}(z).0
\]

may be typed in an environment

\[
\Gamma = \{ b : T, a : T^b, x : ((T)^b)^b \}
\]

for some type \( T \). The second agent in this composition, \( x(y : (T)^w) \mathcal{G}(b).0 \), may synchronise with \(!\mathcal{P}(a).0\) and get the channel \( a \), which thus instantiates \( y \). The question is in which sense can we say that the agent \( x(y : (T)^w) \mathcal{G}(b).0 \) above is only given the capability to write to a channel \( a \). The third term of the composition,

\[
x(y : (T)^b) y(z : T) \mathcal{G}(z).0
\]

must be given the capability both to write to and read from \( a \). But the agent \(!\mathcal{P}(a).0\) cannot be said to be able to pass different capabilities at different times. In fact, if the channel \( a \) is never used in an output action, for instance in

\[
!\mathcal{P}(a).0 \mid x(y : (T)^w) \mathcal{G}(b).0 \mid x(y : (T)^b) \mathcal{G}(b).0,
\]

we could give \( x \) the more restricted sort \( x : ((T)^w)^b \) in an otherwise identical environment in which the channel \( a \) still has the type \( (T)^b \). In fact, nothing prevents us from distinguishing channel types for input actions and output actions. Thus, a channel may be given the type \( (T_i, T_o) \), where \( T_i \) is the type for input actions, and \( T_o \) for input actions. The only requirement is that the type \( T_i \) is more restricted than the type of \( T_o \), i.e. that \( T_i \geq T_o \). In this case, in the expression above \( x \) could be given the more general type \( ((T)^w)^b, ((T)^w)^b \), instead of \((T)^w)^b \), which corresponds to the type \( ((T)^w)^b, ((T)^w)^b \). This gives a more liberal typing, i.e. a typing in which more processes may typecheck, and is also the scheme adopted by Yoshida and Hennessy [998]. This example illustrates the fact that capabilities are not passed at all in a strict sense. What the typing of an agent \( x(y : (T)^w), P \) says about the type of the parameter \( y \) is that it does not occur in input position at \( P \), nor as the object of an output action \( w(y) \) where channel \( w \) cannot be given a type \( (S)^b \) such that \( S \leq (T)^w \).

Output actions are not annotated by typing assertions, in contrast to input actions. Typing rules are thus enforced mainly by typing input actions in a given type.
environment. Typing assertions in input actions of form \(x(y : T).P\) may be viewed as an assertion about the continuation \(P\). This is the most common way of enforcing a typing discipline. The type of value \(z\) in an output action of form \(\pi(z)\) must be no more restrictive than \(T\), i.e. if \(z : S\) then \(T \geq S\), in order to guarantee well-typing. If the type of a channel in a given environment and expression were to be regarded as a capability, the type of a channel should be associated with output actions rather than input actions. The fact that typing assertions must be attached to input actions has consequences for the extension of typing to processes and higher-order communication in Yoshida and Hennessy [187]. This will expanded below.

The system defined above could be described in terms of polarised blocking, without any notion of type, as follows:

\[
(p)(P\mid p^+ | C_1 \mid p^- | C_2 \mid p^-)
\]

Note the relation between capability and blocking. In the typed calculus, within \(P\) the channel \(p\) has type \((T)^r\), i.e. it may only be used for input. In our version, blocking \(P\) with \(p^+\) will guarantee that \(p\) is used only for input, at least for agents in the environment of the printer, which is exactly what is desired. Likewise, agents interacting with \(C_1\) or \(C_2\) may only see output actions along \(p\) from these agents. As a result, \(C_2\) will not be able to disrupt the protocol involving \(C_1\) and the printer. Also, agents in the environment of the system may be able to see both input and output actions along \(p\) if this channel is extruded, in accordance with the sort of \(p\) for the whole system, \((T)^h\).

The main distinction between these approaches lies in the fact that no capabilities are attached to channels except those given by the name of the channel. This conforms to our basic tenet that names in process calculi are pure, i.e. “names commit us to nothing” [123]. Furthermore, in a distributed open environment it is not clear how capabilities might be attached to names, apart from the capabilities associated with the possession of a name.

A typing that defines exactly the kind of processes that cannot communicate along a certain channel \(a\) must be undecidable, since otherwise we could decide whether a process can communicate along any given channel. This set of processes may be defined using the notion of blocking. It consists of all processes \(Q\) such that \(Q \simeq Q\mid a\). As a result, this equivalence is not decidable. Nevertheless, it may be seen as a type definition for the set of processes desired.

Different typing approaches may thus result in more or less processes being typable. As we will see below, a more refined definition of channel typing, presented by Hennessy and Riely [64], allows more processes to typecheck. However, the most serious objection that can be raised against the typing approach concerns its adequacy to open systems. One reason is that the typing of a process may depend on the type of the environment. We do not know if the process

\[P = x(y)\pi(u).P\]

can execute an output action along a certain channel \(z\). In order to know this the environment must be determined. If the environment \(E\) does not have the capability to use \(z\) for output, then the output action \(\pi(z)\) cannot occur in \(E\), otherwise \(P \mid E\) will not typecheck. We have thus a compositional issue here which is hard to solve in the framework of an open system. With blocking we would simply define \(P\) as

\[P = x(y)\pi(u).P \mid z^+\]

This reflects the fact that in open systems it is more straightforward to rely on dynamic monitoring of processes to enforce resource access restrictions, than on static typechecking.
4.6 The $\lambda\pi_v$ Calculus

In Yoshida [187], types are assigned to processes in a higher-order setting, the $\lambda\pi_v$ calculus. The type of a process corresponds to its interface, which determines the resources to which it has access. This kind of typing is expected to limit resource access by controlling the effect of mobile code on local environments. The main result is a Type Safety Theorem stating that no well-typed process can input higher-order code that does not conform to the interface of the process.

The $\lambda\pi_v$ calculus is a call-by-value, call-by-name calculus augmented with $\pi$-calculus primitives. Arbitrary abstractions, including process abstractions, may be sent as values, and also applied as in the $\lambda$-calculus.

Processes in $\lambda\pi_v$ allow judgements of the form

$$\Gamma \vdash P : [\Delta]$$

where $\Delta$ may be seen as a finite name environment mapping names to capabilities. The judgement may be understood as indicating that the process $P$ may use at most the resources in the domain of $\Delta$, which can be seen as an approximation “from above” to $P$’s interface, i.e. $P$ may use at most the resources specified in the domain of $\Delta$.

From now on we will follow the notation for process and types given in Yoshida [187].

4.6.1 Types

The types used in $\lambda\pi_v$ are given in Figure 4.1. Processes may have either type proc, the most general process type with no information about its interface, or $[\Delta]$, where $\Delta$ is a name environment, ranged over by $\pi$. A set of base types is assumed, as well as an infinite set of channel names $N$ ranged over by $a, b, \ldots$, and an infinite set of variables $V$ ranged over by $x, y, \ldots$ (sometimes written as $X, Y, \ldots$, if they express higher order values). An environment $\Delta$ may be thus defined as a mapping from $N \cup V$ to channel types. Channel types take the form $(S, S)$, where $S$ denotes the input sort and $S$ the output sort. Two constants $\bot$ and $\top$ denote the least resp. the highest capability. Values are of three kinds, base types, channel types, and HO-types ranged over by $\sigma_H$. The latter can be either the functional type constructor $\sigma_H \to \rho$, or $(x : \sigma) \to \rho$, where $\rho$ is either an HO-value type or a process type. The type $(x : \sigma) \to \rho$, where $\sigma$ is a channel type, is a dependent functional type. The type $\rho$ is allowed to contain occurrences of the functional variable $x$. This kind of functional type is necessary to assign a process type or interface to the process yielded by the application of an agent $\lambda(x : \sigma).P$, an abstraction on a channel type $\sigma$, to a channel of the same type.

A process $P$ is abstracted by being turned into a thunked process, i.e. a value of form $\lambda($unit$)P$, briefly $(P)$, and type unit $\to [\Delta]$, briefly $(\Delta)$, where $\Delta$ denotes the interface of $P$.

4.6.2 Syntax

The syntax for terms is shown in Figure 4.2. Values consist of basic values and abstractions as in the $\lambda$-calculus. Besides values, application and literals, the other terms are those of the $\pi$-calculus. The definitions of free names and free variables for types and terms are standard, see Yoshida [187] for details. Note that the occurrence of the name $x$ in a dependent type of form $(x : \sigma) \to \rho$ is binding.
**Type**

\[
\begin{align*}
\text{Term:} & \quad \rho := \pi \mid \sigma_H \\
\text{Base:} & \quad \sigma_H := \text{unit} \mid \text{nat} \ldots \\
\text{Process:} & \quad \pi := [\Delta] \mid \text{proc} \\
\text{HOValue:} & \quad \sigma_H := \sigma_H \rightarrow \rho \mid (x : \sigma) \rightarrow \rho \\
\text{Channel:} & \quad \sigma := (S_t, S_o) \\
\text{Value:} & \quad \tau := \sigma_H \mid \sigma \\
\text{Sort:} & \quad S := (\tau_1, \ldots, \tau_n) \mid \top \mid \bot
\end{align*}
\]

**Abbreviation**

\[
\begin{align*}
\text{input only:} & \quad S^i \overset{\text{def}}{=} \{S, \bot\} \\
\text{output only:} & \quad S^o \overset{\text{def}}{=} \{\top, S\} \\
\text{input/output:} & \quad S^{in} \overset{\text{def}}{=} \{S, S\} \\
\text{thunk type:} & \quad \langle \Delta \rangle \overset{\text{def}}{=} \text{unit} \rightarrow [\Delta]
\end{align*}
\]

**Environment**

\[
\begin{align*}
\text{Channel:} & \quad \Delta := \emptyset \mid \Delta, u : \sigma \\
\text{General:} & \quad \Gamma := \emptyset \mid \Gamma, x : \tau \mid \Gamma, a : \sigma
\end{align*}
\]

Figure 4.1: Types for the \(\lambda\pi_v\) Calculus

---

**Term**

\[
\begin{align*}
P, Q, \ldots : & \overset{\text{value}}{=} V \\
& \quad | \overset{\text{nil}}{=} 0 \\
& \quad | P \mid P \\
& \quad | u(V_1, \ldots, V_n)P \\
& \quad | u? (x_1 : \tau_1, \ldots, x_n : \tau_n)P \\
& \quad | uP \\
& \quad | (\nu a : \sigma)P \\
& \quad | PP
\end{align*}
\]

**Identifier**

\[
\begin{align*}
\overset{\text{variable}}{=} u, v, w, \ldots : = l \mid x, y, z \ldots \\
\overset{\text{channel}}{=} a, b, c, \ldots \\
\overset{\text{identifier}}{=} V, W, \ldots := u, v, w, \ldots
\end{align*}
\]

**Value**

\[
\begin{align*}
\overset{\text{replicator}}{=} \lambda(x : \tau)P \\
\overset{\text{abstraction}}{=} \lambda(x : \tau)P
\end{align*}
\]

**Literal**

\[
\begin{align*}
l, l', \ldots : & \overset{\text{unit}}{=} () \mid 1, 2, 3, \ldots \\
\overset{\text{number}}{=} \overset{\text{thunk}}{=} \overset{\text{run}}{=} \overset{\text{run}}{=} \overset{\text{run}}{=}
\end{align*}
\]

**Abbreviations**

\[
\begin{align*}
\overset{\text{def}}{=} \lambda(x : \text{unit})P \\
\overset{\text{def}}{=} \lambda(x : \text{unit})P \\
\overset{\text{def}}{=} \lambda(x : \text{unit} \rightarrow \pi)x() \overset{\text{run}}{=}
\end{align*}
\]

Figure 4.2: Syntax for the \(\lambda\pi_v\) calculus

71
(Reduction)

- \( \beta \) \((\lambda x : \tau)P)V \rightarrow P[V/x] \) (app₁)
- \( Q \rightarrow Q' \)
- \( PQ \rightarrow PQ' \) (app₂)
- \( P \rightarrow P' \)
- \( PV \rightarrow PV' \)

(com) \( a?(x_1 : \tau_1, \ldots, x_n : \tau_n)P \mid a!(V_1, \ldots, V_n)Q \rightarrow P[v_1, \ldots, V_n/x_1, \ldots, x_n] \mid Q \)

(par) \( P \rightarrow P' \)

(str) \( P \equiv P' \rightarrow Q' \equiv Q \)

(Structural Equivalence)

- \( P \equiv Q \) if \( P \equiv Q \)
- \( P \mid Q \equiv P \mid Q \) \( (P \mid Q) \mid R \equiv P \mid (Q \mid R) \) \( P \mid 0 \equiv 0 \) \( *P \equiv *P \)
- \( \langle \nu : \sigma \rangle 0 \equiv 0 \) \( \langle \nu : \sigma \rangle P \mid Q \equiv \langle \nu : \sigma \rangle (P \mid Q) \) if \( a \notin \text{fn}(Q) \)
- \( \langle \nu : \sigma \rangle \langle \nu b : \sigma' \rangle P \equiv \langle \nu b : \sigma' \rangle \langle \nu : \sigma \rangle Q \) if \( a \notin \text{fn}(\sigma') \) and \( b \notin \text{fn}(\sigma) \)

Figure 4.3: Reduction for the \( \lambda \pi_v \) calculus

\[ \begin{align*}
\text{T}{\nu/x} & = \text{T} \quad \text{⊥}{\nu/x} = \text{⊥} \quad \sigma_B{\nu/x} = \sigma_B \quad \text{proc}{\nu/x} = \text{proc} \\
\langle S, S \rangle{\nu/x} & = \langle S, S \rangle{\nu/x} \quad \langle S, S \rangle{\nu/x} \\
(\tau_1, \ldots, \tau_n){\nu/x} & = (\tau_1{\nu/x}, \ldots, \tau_n{\nu/x}) \\
(\sigma_B \rightarrow \rho){\nu/x} & = \sigma_B{\nu/x} \rightarrow \rho{\nu/x} \\
(\nu : \sigma \rightarrow \rho){\nu/x} & = (\nu : \sigma{\nu/x}) \rightarrow \rho{\nu/x} \text{ with } x \neq y \\
[\Delta]{\nu/x} & = \bigcup[w{\nu/x} : \sigma{\nu/x}] \text{ with } w : \sigma \in \Delta
\end{align*} \]

Figure 4.4: Name substitution into types for the \( \lambda \pi_v \) calculus

4.6.3 Semantics

The reduction semantics is given in terms of a structural equivalence, denoted \( \equiv \),
and a binary relation

\[ P \rightarrow Q \]

between terms which contain no free variables, also called \emph{programs}. The formal
definitions are given in Figure 4.3. Value substitution into terms \( P{\nu/x} \) is defined
inductively on the structure of terms. Note that values are also substituted into
types in the rules \( \beta \) and \( \text{(com)} \). Since only channel names and variables occur
in these types, the definition of \( \rho{\nu/x} \) is the identity unless \( \nu \) is a channel name or
variable. The definition of name substitution into types is given in Figure 4.4.

The operator \( \sqcup \) acts on types and denotes the least upper bound with respect to a
subtyping order on the types. This least upper bound may be seen intuitively as the
union of the accessibility rights of two processes. It may be shown that in properly
typed environment the operator \( \sqcup \) is always defined. For details see Yoshida[187].

4.6.4 Well-formed Types and Environments

Three form of judgements are defined:
The inference rules are given in Figure 4.5. The rules for type formation are constrained to identifiers which are already declared in the current environment. For further details we refer to Yoshida [187]. The definition of the restriction rule (RES) is formulated in terms of an erasure operator which is defined in Figure 4.6.

4.6.5 Type Inference

Judgements are of the form

\[ \Gamma \vdash u : \sigma \quad \text{a name } u \text{ has type } \sigma \text{ under the environment } \Gamma \]
\[ \Gamma \vdash P : \alpha \quad \text{a term } P \text{ has type } \alpha \text{ under the environment } \Gamma. \]

The inference rules are given in Figure 4.7. The \textbf{(Function)} rules are inherited from typing systems for the polymorphic \( \lambda \)-calculus, whereas the \textbf{(Process)} rules are based on IO-typing systems [134, 64, 188]. Note specially the presence of abstractions over channel variables in the rule (ABS\(_N\)), and the corresponding elimination rule (APP\(_N\)), which allows dynamic instantiation into types during \( \beta \)-reduction.

<table>
<thead>
<tr>
<th>Well-formed Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e-nil) ( \emptyset \vdash \text{Env} ) (e-val) ( \Gamma \vdash \tau : \text{tp} \quad u \notin \text{dom}(\Gamma) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Well-formed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t-base) ( \Gamma \vdash \text{Env} ) ( \Gamma \vdash \tau : \text{tp}, \bot : \text{tp}, \sigma_B : \text{tp}, \text{proc} : \bot : \text{tp} )</td>
</tr>
</tbody>
</table>

\[ \left( t = \text{abs}_H \right) \quad \Gamma \vdash \sigma_H : \text{tp}, \rho : \text{tp} \]

\[ \left( t = \text{abs}_N \right) \quad \Gamma, x : \sigma \vdash \rho : \text{tp} \]

\[ \left( t = \text{proc} \right) \quad \forall \alpha \in \text{dom}(\Delta), \Gamma \vdash \Gamma(u) \leq \Delta(u) \]

\[ \left( t = \text{chan} \right) \quad \Gamma \vdash S_i \geq S_i \]

\[ \left( s = \text{id} \right) \quad \Gamma \vdash \alpha : \text{tp} \quad (s = \text{sort}) \quad \Gamma \vdash \tau_i : \text{tp} \rightarrow \text{tp} \]

\[ \left( s = \text{base} \right) \quad \Gamma \vdash [\Delta] : \text{tp} \quad \left( s = \text{chan} \right) \quad \Gamma \vdash \tau_i \leq (\tau_1, \ldots, \tau_n) \leq \tau \]

\[ \left( s = \text{proc} \right) \quad \Gamma \vdash \tau_i \leq \rho \leq \sigma_H \rightarrow \rho' \]

\[ \left( s = \text{proc} \right) \quad \Gamma \vdash [\Delta] : \text{tp} \]

Figure 4.5: Well-formed Types and Subtyping for the \( \lambda \pi_c \) calculus

\[ \Gamma \vdash \text{Env} \quad \Gamma \vdash \alpha : \text{tp} \quad \alpha \text{ is a well-formed type in the environment } \Gamma \]
\[ \Gamma \vdash \alpha \leq \alpha' \quad \alpha \text{ is less than } \alpha' \text{ in the environment } \Gamma \]

The formal system is given in Figure 4.5. The rules for type formation are constrained to identifiers which are already declared in the current environment. For further details we refer to Yoshida [187]. The definition of the restriction rule (RES) is formulated in terms of an erasure operator which is defined in Figure 4.6.
\[
\Gamma /a = \top, \bot/a = \bot, \sigma_B/a = \sigma_B, \text{proc} /a = \text{proc}
\]
\[
\langle S_1, S_2 \rangle /a = \langle S_1/a, S_2/a \rangle
\]
\[
\langle \tau_1, \ldots, \tau_n \rangle /a = \langle \tau_1/a, \ldots, \tau_n/a \rangle
\]
\[
\sigma /a = \sigma /a \rightarrow \rho/a
\]
\[
(\langle X : \sigma \rangle \rightarrow \rho) /a = (\langle X : \sigma/a \rangle \rightarrow \rho/a)
\]
\[
[\Delta] /a = \{ u : (\sigma/a) \mid u : \sigma \in \Delta \land u \neq a \}
\]

Figure 4.6: Name Erasing from Types for the \(\lambda\pi_v\) calculus

<table>
<thead>
<tr>
<th>(Common)</th>
<th>(VAL)</th>
<th>(CON)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\Gamma, u : \tau, \Gamma \vdash \text{Env} )</td>
<td>(\Gamma \vdash \text{Env} )</td>
</tr>
<tr>
<td></td>
<td>(\Gamma, u : \tau, \Gamma \vdash u : \tau )</td>
<td>etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(SUBn)</th>
<th>(SUBn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash P : \rho )</td>
<td>(\Gamma, \rho \leq \rho' )</td>
</tr>
<tr>
<td>(\Gamma \vdash P : \rho' )</td>
<td>(\Gamma \vdash u : \sigma \leq \sigma' )</td>
</tr>
<tr>
<td>(\Gamma \vdash u : \sigma' )</td>
<td>(\Gamma \vdash u : \sigma' )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(Function)</th>
<th>(APPN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma, X : \sigma_B \vdash P : \rho )</td>
<td>(\Gamma \vdash P : \sigma_B \rightarrow \rho )</td>
</tr>
<tr>
<td>(\Gamma \vdash Q : \sigma_B )</td>
<td>(\Gamma \vdash PQ : \rho )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(ABSn)</th>
<th>(APPN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma, x : \sigma \vdash P : \rho )</td>
<td>(\Gamma \vdash P : (x : \sigma) \rightarrow \rho )</td>
</tr>
<tr>
<td>(\Gamma \vdash \lambda(x : \sigma)P : (x : \sigma) \rightarrow \rho )</td>
<td>(\Gamma \vdash u : \sigma )</td>
</tr>
<tr>
<td>(\Gamma \vdash Pu : \rho[u/x] )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(Process)</th>
<th>(NIL)</th>
<th>(PAR)</th>
<th>(REP)</th>
<th>(RES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash \text{Env} )</td>
<td>(\Gamma \vdash P_1 : \pi )</td>
<td>(\Gamma \vdash P_2 : \pi )</td>
<td>(\Gamma \vdash P : \pi )</td>
<td>(\Gamma, a : \sigma \vdash P : \pi )</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{NIL} )</td>
<td>(\Gamma \vdash P_1 : \pi )</td>
<td>(\Gamma \vdash P_2 : \pi )</td>
<td>(\Gamma \vdash \text{PAR} )</td>
<td>(\Gamma \vdash (\nu a : \sigma)P : \pi/a )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(OUT)</th>
<th>(IN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\pi \vdash u : (\tau_1, \ldots, \tau_n)^0 )</td>
<td>(\pi \vdash u : (\tau_1, \ldots, \tau_n)^1 )</td>
</tr>
<tr>
<td>(\Gamma \vdash V_1 : \tau_1 \Rightarrow \pi \vdash V_i : \sigma_i )</td>
<td>(\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash P : \pi, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash P : \pi )</td>
</tr>
<tr>
<td>(\Gamma \vdash u(V_1, \ldots, V_n) : \pi )</td>
<td>(\Gamma \vdash u(x_1) : (x_1 : \tau_1, \ldots, x_n : \tau_n)P : \pi )</td>
</tr>
</tbody>
</table>

\[\pi \vdash u : \sigma \quad \text{def} \quad \Gamma \vdash [u : \sigma] \leq \pi\]
\[\pi, x : \sigma \quad \text{def} \quad \pi \sqcup [x : \sigma]\]
\[\pi, x : \sigma _H \quad \text{def} \quad \pi\]

Figure 4.7: Typing system for the \(\lambda\pi_v\) calculus
4.6.6 Type Soundness

The main result of the type system of the $\lambda\pi$ language is a Subject Reduction Theorem, stating the following:

If $\Gamma \vdash P : \rho$ and $P \rightarrow P'$, then $\Gamma \vdash P' : \rho$.

The judgement $\Gamma \vdash P : \Delta$ means that $P$ can use at most the resources mentioned in $\Delta$ according to the capabilities that these resources are assigned in $\Delta$. This idea is formalised by the definition of a unary predicate $\Gamma \vdash P : \pi \rightarrow \text{err}$, stating basically that relative to $\Gamma$, $P$ violates the interface $\pi$ if it can input on a channel $a$ to which the interface $\pi$ does not assign any input capability, and analogously for output.

Finally, a type safety result states that

If $\Gamma \vdash P : \pi$ then $\Gamma \vdash P : \pi \rightarrow \text{err}$

4.6.7 An Example

We illustrate with an example from Yoshida [187]. A process of the form

$c?(x : \Delta_{ab}) \text{run } x$

indicates that it is only willing to accept processes for execution that respects the interface denoted by $\Delta_{ab}$, defined as

$$\Delta_{ab} = \{a : (\text{int})^1, b : (\text{int})^0\}$$

Thus, a process of type $\Delta_{ab}$ will be able only to input a value of type $\text{int}$ along channel $a$, and output a value of type $\text{int}$ along $b$. The consistency of local resources is maintained in principle by the fact that imported code instantiating the variable $x$ in $c?(x : \Delta_{ab}) \text{run } x$ is constrained by the interface $\Delta_{ab}$.

Process types may be nested. Let $\langle \Delta_a \rangle$ and $\langle \Delta_c \rangle$ denote the types $\langle a : \text{int}^0 \rangle$ resp. $\langle c : \langle \Delta_a \rangle^0 \rangle$. Then the process

$$\ast\text{req?}(y : \langle \Delta_c \rangle)(\text{run } y \mid c?(x : \langle \Delta_a \rangle)(\text{run } x \mid a?(z : \text{int}P))$$

may download on the request channel req a process that can only send along $c$ a process abstraction which, in turn, can only send values of type $\text{int}$ along $a$.

Dependent types are also introduced. For instance, a process abstraction $Fw$, which is called forwarder in Yoshida [187], is defined as follows:

$$Fw \overset{\text{def}}{=} \lambda x y (\ast x?(z : \text{int})y!(z))$$

where $\ast P$ denotes the replication of the process $P$. The forwarder is the abstraction of a process that repeatedly inputs some value of a type $\text{int}$ on channel $x$ and outputs it immediately on $y$. After instantiation, for instance by applying the forwarder to channels $a$ and $b$, $Fw(ab)$, we obtain a process whose interface may be described as $\Delta_{ab}$. The abstraction $Fw$ is given the dependent type

$$x : (\text{int}^1) \rightarrow (y : (\text{int}^0) \rightarrow [x : (\text{int}^1), y : (\text{int}^0)])$$

which represents a dependent type that after instantiation of $x$ and $y$ will result in the process type $[x : (\text{int}^1), y : (\text{int}^0)]$.

Consider now the following process:
\[ c?z : (\text{int}^0) z!1.0. \]

This process takes along channel \( c \) a channel \( z \) of type \( (\text{int}^0) \), i.e. a channel that may be used to output a value of type integer. What is the type of this agent? According to the typing rules of \( \lambda \pi_e \), it has type \( [\Delta_c] \) where

\[ \Delta_c \overset{\text{def}}{=} \{ c : (\text{int}^0)^1 \} \]

It is obvious that this process may gain more capabilities by receiving an input on \( c \). The solution is to let the process sending the new capability be given an extended interface that takes account of the capability passed along the channel. Thus, a process of form

\[ c!d \]

is typed as \( \Delta_{cd} \), defined as

\[ \Delta_{cd} \overset{\text{def}}{=} \{ c : (\text{int}^0)^1, d : (\text{int})^0 \} \]

Obviously, \( d \) is not in the interface of \( c!d \). This illustrates once again the limitations of capability typing. A process that may receive new information by input cannot be adequately typed without knowledge of the environment in which it takes part, which goes against the grain on open systems. Also, the claim that the interface of the process restricts the resources to which it has access is contradicted hereby. For instance, the process

\[ a(X : \langle \Delta_c \rangle) \text{ run } X \]

may receive along \( a \) a process thunk of type \( X : \langle \Delta_c \rangle \). However, the interface \( \Delta_{c2} \) assigned to the parameter \( X \) certainly does not restrict the resources to those accessible on the channel \( c \) alone, and as a result the ability to constrain the effect of imported code is lost.

Worse yet, it may be necessary to declare, in the interface of an imported code, all the reply channels which may be imported. We illustrate this with the following example, also taken from Yoshida [187].

A computer server is defined as follows:

\[ \text{Server}(\text{req}, s, p) \Leftarrow \text{req}?(X)Xsp \]

This server takes a script \( X \), a process abstraction parameterised on two service ports, and applies it to the actual service ports \( s \) and \( p \). The service port \( s \) gives access to a server \( \text{Succ}(a) \), defined by

\[ \text{Succ}(a) \Leftarrow \ast a?(y, z)z!(\text{succ}(y)) \]

This represents a service situated at \( s \) which receives a value of type integer resp. a reply channel, and returns along the reply channel the successor of the value received.

The service port \( p \) gives access to a service defined by

\[ \text{Pred}(a) \Leftarrow \ast a?(y, z)z!(\text{pred}(y)) \]
i.e. a process that takes a value \( y \) of type integer and a reply channel \( z \), and returns along the latter the predecessor of the received value.

Two examples of clients are the following. Client (A) wants to increment a number \( k \) twice. It may be defined by

\[
\text{Client}_A(\text{req}) \triangleright= \text{req}\langle \lambda(s,p)((uc)s!(k,c)z!s!(z,c)\text{Fw}(\sigma_A)) \rangle
\]

Client (B), on the other hand, wants to evaluate the successor and the predecessor of two different numbers:

\[
\text{Client}_B(\text{req}) \triangleright= \text{req}\langle \lambda(s,p)(vcc')((s!(c\;r_{1B}) | p!(m,c')\text{Fw}(c'\;r_{2B})) \rangle
\]

As we may observe, final results are relayed on the result channels, \( r_A \) for client (A), and \( r_{1B}, r_{2B} \) for client (B).

The system may now be described by

\[
\text{Client}_A(\text{req}) | \text{Client}_B(\text{req}) | \text{Server}(\text{req},s,p) | \text{Succ}(s) | \text{Pred}(p)
\]

The question now is how the server is going to be typed. If we define \( \Delta_r \) by

\[
\Delta_r \overset{\text{def}}{=} [r_A : (\text{int}^0), r_{1A} : (\text{int}^0), r_{1B} : (\text{int}^0)]
\]

and \( \sigma^0_s \) as the type \((\text{int},(\text{int}^0)^0)\), then we may type the body of Client (A), \( P_A \), defined as

\[
P_A \overset{\text{def}}{=} (uc)s!(k,c)z!s!(z,c)\text{Fw}(\sigma_A),
\]

by

\[
P_A : [s : \sigma^0_s, p : \sigma^0_p, \Delta_r]
\]

The type of the abstraction of the body of client (A) becomes thus

\[
\lambda(s : \sigma^0_s, p : \sigma^0_p) . P_A : \tau_{sc}
\]

where \( \tau_{sc} \) is the type of the script:

\[
\tau_{sc} \overset{\text{def}}{=} (s : \sigma^0_s) \rightarrow (p : \sigma^0_p) \rightarrow [s : \sigma^0_s, p : \sigma^0_p, \Delta_r]
\]

If we now let \( \Delta_d \) denote the environment \( \{\text{req} : (\tau_{sc})^0, \Delta_r\} \), \( \text{Client}_A(\text{req}) \) as well as \( \text{Client}_A(\text{req}) \) may be typed by \( [\Delta_d] \).

In order to type the server, we need to add the ports \( s \) and \( p \) at the interface, together with their types. Thus, we may define the type of \( \text{Server}(\text{req},s,p) \) as \([\Delta_{serv}]\), where

\[
\Delta_{serv} \overset{\text{def}}{=} \{\text{req} : (\tau_{sc})^1, \Delta_r, s : \sigma^0_s, p : \sigma^0_p \}
\]

This means that the parameter \( X \) in the definition of the server must be given type \( \tau_{sc} \), and the server ought to be defined as

\[
\text{Server}(\text{req},s,p) \triangleright= \text{req}\langle (X : \tau_{sc})Xsp \rangle
\]

Hence, the type of the parameter denoting the received script in the server must all the possible reply channels that agents in the environment may use. Apart from the fact that these channels are commonly defined as fresh names, and may
not be known by the server, we must assume here also that the environment is closed, and that the number of possible reply channels is finite. This is clearly unsatisfactory, and lends further evidence to the fact that in the context of open systems, typing of capabilities is not adequate as a way of specifying resource access restrictions.

These problems are partly a consequence of the fact, mentioned in a previous section, that type annotations must be attached to input actions, which is by its turn a consequence of the fact that process and channel typing has less to do with capabilities in the strict sense of the word, than with a description of behaviour. Typing higher order input parameters is the only way of specifying a capability restriction, meaning that only agents whose interface is at least as restrictive as the declared type can be accepted. In the case of first-order input, i.e. input of channels, the situation is distinct, since channels cannot be given an interface. The solution is to rely on the environment for correctly typing the continuation of a first-order input action. In general, the limitations associated with process typing are closely related to the complexities involved in channel input. As long as channel input is allowed, a feature that allows extension of the capabilities possessed by an agent, the only way to fully specify the interface of an agent is by “closing” the environment, i.e. by fully specifying it. However, in an open systems framework this is not possible. That is the main reason, in our opinion, why access rights and capabilities are enforced basically by dynamic monitoring, not by typing. The behaviour of any process that is not wholly or almost wholly isolated from the environment, is always extensible in a way that may be impossible to check statically. The alternative is to enforce safety restrictions at runtime.

4.6.8 Encoding Protection Domains with Types

We discuss here the difficulties involved in encoding the example shown in Section 4.4.3. In that example, a site accepts incoming scripts that are executed in a protection context according to the privileges assigned to the requesting party. The principal, called Domain, accepts along channel d a pair of arguments consisting of the name of the group to which the requesting party belongs, and the script that should be executed. An access control policy for the received script is then elected in accordance with the first argument.

This scheme is not easy to encode in a typed calculus. The reason is that the channel d along which requests are passed must be assigned a type that is as general as possible in order to be well-typed. It is not clear in this context how the access control policy could be enforced. A rather awkward solution would be to dedicate one channel to each access control group. In this case the access requirements would be enforced by the type of the argument instantiating to scripts. For instance, if we let $T_1$, $T_2$ and $T_3$ denote the interface type associated with access control policies $a_1$, $a_2$ resp. $a_3$, we can dedicate one channel $d_1$, $d_2$ resp. $d_3$ for each interface type as follows:

\[
\text{Domain} \overset{def}{=} \langle *d_1(\text{Script} : T_1) \text{run Script} | *d_2(\text{Script} : T_2) \text{run Script} | *d_3(\text{Script} : T_3) \text{run Script} | \text{resources} \ldots \rangle
\]

This implies that knowledge of the channels $d_1$, $d_2$ and $d_3$ may be seen as a capability given an agent the power to submit any scripts with the corresponding interface. Well-typing assumes that agents that have acquired the capability are well-behaved.
Furthermore, all possible reply channels must be defined in the corresponding type $T_i$, and no channels except those naming resources are allowed to have type $T^I$ or a subtype of it, since this would imply that the script may acquire additional capabilities through inputs that violate the privileges assigned to the script.

Note that it is not possible to define the process as follows:

$$\text{Domain} \overset{\text{def}}{=} (\ast d(\text{group, Script : proc}).$$

$$\left[ \begin{array}{l}
[\text{group} = a_1][d_1](\text{Script}) \\
[\text{group} = a_2][d_2](\text{Script}) \\
[\text{group} = a_3][d_3](\text{Script}) \\
\end{array} \right]$$

$$d_1?\text{Script : T}_1) \text{run Script}$$

$$d_2?\text{Script : T}_2) \text{run Script}$$

$$d_3?\text{Script : T}_3) \text{run Script}$$

resources...)

In this version of the principal, channel $d$ receives the name of a group and a script, and distributes the scripts along channels $d_i$ according to the value of the first argument, which should be instantiated to the name of one of the groups. This would assure that processes are well-typed according to types $T_i$, but in reality this agent is not typable. The reason is that the type of $d_i$ should be at least as large as $(\text{proc})^m$, where proc is the largest process type. In general it is not possible to declare an agent which receives values of a certain general type $T$ along a given channel, and then distributed these values according to the subtype of $T$ to which they may belong.

Another, more elegant, but scarcely more flexible solution would be to let channel $d$ be type as $d : T$ where $T$ corresponds to the type of an abstracted agent $\lambda(\bar{x} : \bar{\tau}).P$, where $\bar{x}$ denotes the names of all available resources, and $\bar{\tau}$ their corresponding types. This would not eliminate the need to include in the types $\bar{\tau}$ the names and types of all reply channels, and to impose the same severe restrictions on communications on the agent $P$. Moreover, this assumes compliance with a detailed protocol specifying the order and types of the access channels of all resources included in the principal. The principal could now be defined as

$$\text{Domain} \overset{\text{def}}{=} (\!(d(\text{group, Script : T}).$$

$$\left[ \begin{array}{l}
[\text{group} = a_1](\text{Script} \langle \bar{b}_1 \rangle) \\
[\text{group} = a_2](\text{Script} \langle \bar{b}_2 \rangle) \\
[\text{group} = a_3](\text{Script} \langle \bar{b}_3 \rangle) \\
\end{array} \right]$$

resources...)

Thus, the process abstraction instantiating the argument Script is applied to a channel vector $\bar{b}_i$ denoting the names of the available resources. Resources whose access is denied may be given a faked channel name in $\bar{b}_i$, e.g. a restricted name without any correlates. Attempts to communicate along these channels would simply block.

What seems clear here is the fact that typing imposes very severe restrictions on the way agents may be defined, yielding awkward encodings and a very inflexible framework for studying protection systems. Furthermore, the types assigned to scripts depend on the types of other agents in the environment. In fact, the only agents whose types are of interest here are the scripts, which should be assigned
types that are not conditioned on the type of the environment, among other reasons because the environment should be assumed to be open and thus not typable a priori.

4.7 The $\lambda \pi_\nu \beta$ Calculus

In order to compare the static typing protection scheme with dynamic monitoring via the filter operator, and to show how type constraints can be encoded in terms of filtering, we present here what we call the $\lambda \pi_\nu \beta$ calculus. This calculus is basically a simplified form, in terms of typing, of the $\lambda \pi_\nu$ calculus, extended with the filter operator. We don’t abandon types altogether, since in a higher-order context typing may express other constraints than those related to capabilities. The resulting type scheme is however much simpler, and dependent types are no longer necessary. The main thrust of the translation is to reduce typing constraints to a filtering scheme applied to a process.

In the $\lambda \pi_\nu \beta$ calculus the notion of channel capability is eliminated altogether, as well as that of process type as interface. Dependent types are also eliminated. The only notion of type present is the traditional one based on data abstraction types.

The formal definition of the type system is given in Figure 4.8. All processes are given the general type $\text{proc}$. Consequently, no dependent types are necessary, since a process type does not include any information about its interface. A channel may always be used either for input or output, and its type is defined simply on the basis of the type of the values that may be sent along it.

The syntax of $\lambda \pi_\nu \beta$ includes the filter operator, and is shown in Figure 4.9. Otherwise, it is similar to the syntax of the $\lambda \pi_\nu$ calculus.

Reduction has to be redefined. The new semantics is given in Figure 4.10. We still use the notion of structural equivalence, but have to refine the definition of transition in order to allow action “at a distance.” We thus introduce the notion of labelled reduction in the calculus. The reason for this is basically that, like the blocking operator, the filter operator applied to a process $P$ cannot always be “extruded” to include any agent composed with $P[L]$. Hence, interaction cannot be reduced to “reaction by contiguity” in the chemical style, cf. the rule (com) in Figure 4.3. Nevertheless, by pushing the $\nu$ operator outside, the simplicity of the chemical style semantics may be partly preserved, as the complexity of the labelled
transition style is largely caused by the need to extrude restricted channels, cf. the rules OPEN and CLOSE in Milner, Parrow, Walker [112]. However, proofs based on induction on the derivation of a transition, e.g. in Theorem 1 below, must take this fact into account.

A subtyping relation is not necessary, unless it is introduced at the level of the basic types. Since names do not occur in types, an environment is well-formed simply if the types occurring in it are well-formed, i.e. if they are constructed according to the formation rules for types, and no names or variables have their types defined twice. The rules are given in Figure 4.11. However, note that the requirement that types are well-formed according to the formation rules is not strictly obeyed, since e.g. a type of form proc → τ is well-formed according to the formation rules for types, but no process with this type can be constructed according to the rules of the typing system. Since the syntax for terms does not allow the construction of terms of this type, no damage is done hereby.

The typing system of the λπvβ calculus is given in Figure 4.12, and may be seen as a simplified version of the type system for the λπv calculus.

We have the following results:

Theorem 1: Subject Reduction

If Γ ⊢ P : ρ and P → P′, then Γ ⊢ P′ : ρ

Proof: By induction on the derivation of P → P′, along the lines followed in Yoshida [187] for the proof of the corresponding theorem for the λπv calculus. The proof here is easier because of the much simplified type system. However, we must also establish that subject reduction holds for labelled transitions P ⊢ a P′, which may also be proved by induction on the derivation of the transition.

As in Yoshida [187], we also give a type safety theorem stating that no runtime errors occur in well-typed processes. The idea of runtime error is formalised by a predicate P ⊢ err, defined in Figure 4.13. We obtain the following result:

Theorem 2: Type safety If Γ ⊢ P : proc, then P ⊬ err.
(Reduction)

(input) \( a^?((x_1 : \sigma_1, \ldots, x_n : \sigma_n)P^{a^?((x_1 \ldots, x_n)P} \xrightarrow{P\{V_1/x_1, \ldots, V_n/x_n\}} P \{x_1, \ldots, x_n\} \)

(output) \( a^!((V_1, \ldots, V_n)P^{a^!((V_1 \ldots, V_n)P \xrightarrow{P} P \{V_1, \ldots, V_n\}} P \)

(filt +) \( \frac{P^{b^+((V)P} \xrightarrow{P} P^{b^+((V)P} & b^+ \in L \) (filt -) \( \frac{P^{b^-((V)P} \xrightarrow{P} P^{b^-((V)P} & b^- \in L \)

(par) \( \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q'}{P \mid Q \xrightarrow{\epsilon} P' \mid Q'} \)

(com) \( \frac{P^{a^?((x_1, \ldots, x_n)P'} Q \xrightarrow{P' \{x_1/x_1, \ldots, V_n/x_n\} Q'} P \mid Q \xrightarrow{P'} P' \{x_1/x_1, \ldots, V_n/x_n\} Q' \)

(app, ) \( \frac{Q \xrightarrow{Q} P \mid Q \xrightarrow{P} P \mid Q} \)

(appr) \( \frac{P \xrightarrow{P'} P \mid V \xrightarrow{P'V} P'V} \)

(par) \( \frac{P \xrightarrow{P'} P \mid Q \xrightarrow{P'} P' \mid Q} \)

(res) \( \frac{P \xrightarrow{P'} (\nu \alpha : \sigma)P \xrightarrow{(\nu \alpha : \sigma)P} (\nu \alpha : \sigma)P} \)

(filter) \( \frac{P \xrightarrow{P'} P \mid L \xrightarrow{P' \mid L} P' \mid L} \)

(str) \( \frac{P \equiv P' \quad Q \equiv Q'}{P \mid Q \equiv P \mid Q} \)

(Structural Equivalence)

\( \bullet P \equiv Q \text{ if } P \equiv_n Q. \)

\( \bullet P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad P \mid 0 \equiv P \quad *P \equiv P \quad *P \equiv P \)

\( \bullet (\nu \alpha : \sigma)0 \equiv 0 \quad (\nu \alpha : \sigma)P \mid Q \equiv (\nu \alpha : \sigma)(P \mid Q) \quad \text{if } a \not\in \text{fn}(Q) \)

\( (\nu \alpha : \sigma)(\nu \beta : \sigma')P \equiv (\nu \beta : \sigma')(\nu \alpha : \sigma)Q \quad \text{if } a \not\in \text{fn}(\sigma') \text{ and } b \not\in \text{fn}(\alpha) \)

\( \bullet P[a \equiv Q] a \text{ if } P \equiv Q \)

Figure 4.10: Reduction for the \( \lambda \pi \beta \) calculus

(Well-formed Environment)

\( \bullet \emptyset \vdash \text{Env} \quad (e-\text{nil}) \quad \Gamma \vdash u : \sigma_H : \text{tp} \quad u \not\in \text{dom}(\Gamma) \)

\( \Gamma, u : \sigma_H \vdash \text{Env} \quad (e-\text{val}) \quad \Gamma \vdash \text{Env} \)

(Well-formed Types)

\( (t-\text{base}) \quad \Gamma \vdash \sigma_H : \text{tp}, \text{proc} : \text{tp} \quad (t-sort) \quad \Gamma \vdash \tau : \text{tp} \)

\( \Gamma \vdash (\sigma_1, \ldots, \sigma_n) : \text{tp} \quad (t-\text{abs}) \quad \Gamma \vdash \sigma_H : \text{tp}, \rho : \text{tp} \)

\( \Gamma \vdash \sigma_H \rightarrow \rho : \text{tp} \quad (t-\text{chan}) \quad \Gamma \vdash S : \text{tp} \)

\( \Gamma \vdash (S) : \text{tp} \)

Figure 4.11: Well-formed Types for the \( \lambda \pi \beta \) calculus

82
As in Yoshida [187], by proving that $P \vdash \text{proc}$ implies that $\Gamma \vdash P : \text{proc}$ is not derivable.

### 4.8 Encoding $\lambda \pi_v$ into $\lambda \pi_v/\beta$

The specification of security requirements cannot be immediately deduced from the various type annotations given to an agent in the $\lambda \pi_v$ calculus. There is also no obvious and unequivocal mapping between type annotations and access control requirements. Apparently, in order to encode these requirements in the $\lambda \pi_v/\beta$ calculus we have to interpret them on the basis of the typing annotations and the environment. Since the security requirements are not readily available, several such interpretations might be possible. The one we choose here is based on the type of the parameter of an input action. We interpret these types as a safety specification for resource access. Since there are no explicit indications about which agents represent resources, we assume here that any restriction upon a channel that may be deduced from the type given to it is a safety requirement.

We explain these ideas below by a series of short examples.
4.8.1 Encoding Interfaces in the $\lambda\pi\beta$ Calculus

We begin by analysing the meaning of a process interface in relation to access control requirements. In simple cases the interface of a process might be enforced dynamically by the filter operator. To illustrate this point, we return to an example presented in Section 4.6.7, the process

$$c?(X : \langle \Delta_{ab} \rangle) \text{run } X$$

where

$$\Delta_{ab} = \{a : (\text{int})^1, b : (\text{int})^0\}$$

With the filter operator, this may be encoded as

$$c?(X)(\text{run } X)[\{a^-, b^+\}]$$

The next example is the process with nested types:

$$*\text{req}?(Y : \langle \Delta_i \rangle)(\text{run } Y \mid c?(X : \langle \Delta_a \rangle)(\text{run } X \mid a?(z : \text{int})P))$$

Using the filter operator, this process may be defined as

$$*\text{req}?(Y)(\text{run } Y)[c^+ \mid c(X).\text{run } X][a^+ \mid a(z)P]$$

As we may observe, since no type information is attached to processes, it is the responsibility of the receiving context to enforce the interface policy by judicious use of the filter operator. Therefore, the notion of type nesting has no correlate in our calculus.

A serious difficulty with process types, as explained before, concerns the fact that a process that may receive information from the environment, typically via an input action, cannot adequately be given an interface, since the latter may be extended dynamically and can be thus seen as potentially infinite. However, a restriction policy for a process can always be declared explicitly by the filter operator, independently of the input capabilities of the process. Moreover, in this case a restriction policy for a process can be stated independently of the context in which the process may take part. In our opinion, the description of an access control policy is not adequate if it cannot be defined completely without knowledge of the context.

The technique shown in the example above cannot be generalised to cases where the interface is not complete, as is the case for a process $P$ defined as

$$P \overset{\text{def}}{=} x?(Y : \Delta_{c^?}) \text{run } Y$$

This process cannot be adequately encoded by

$$x?(Y).\text{run } Y[\{c^\}\}$$

since this would mean that the process instantiating $Y$, which could be

$$c?(z : (\text{int})^0)z!(1)$$

can only communicate along $c$, which is not what is expected here. The interface of this process consists rather of input actions along the the channel $c$ and output actions along any channel of type $(\text{int})^0$. If the name environment $\Gamma$ is given the latter may be approximated by the information available in it. An interpretation
of the resource access policy of a process of this kind is that there is no resource of type \((\text{int})^I\) which needs protection, and thus any channel of this type may be allowed to perform output actions without damage. As a result, for protection purposes we can safely encode \(P\) as follows. Assume \(I\) represents the (polarised) channels belonging to the domain of the environment \(\Gamma\) of the well-typed process \(\text{Sys}\) in which \(P\) is a component. This is basically the domain of \(\Gamma\) with polarities, and will be formalised below. The name of any resource that should be protected is assumed to be included in \(I\). Now, \(P\) may be encoded as

\[
x ? (Y). (\text{run } Y) [B]
\]

where

\[
B \overset{\text{def}}{=} \{c^-\} \cup \{a \in I \text{ s.t. } a : (\text{int})^0\}
\]

In other words, we may safely execute the incoming process thunk instantiating the process variable \(Y\) in a context which filters inputs along \(c\) and outputs along any channels in \(I\) of type \((\text{int})^0\). This is the basic idea for the encoding. Nevertheless, the presence of higher order values, other than process thunks, that can be imported via communication poses additional problems for the formalisation of the encoding.

### 4.8.2 Higher-Order Values and Types

An agent which may be the object of a communication action is necessarily an abstraction, either a thunk process, an agent abstracted on a channel, or another kind of higher-order abstraction. For instance, in

\[
c ? (X : \langle \Delta_{ab} \rangle)(\text{run } X)
\]

the type of \(X\) is \(\langle \Delta_{ab} \rangle\), a thunk process with interface \(\Delta_{ab}\), where

\[
\Delta_{xy} \overset{\text{def}}{=} \{x : (\text{int})^I, y : (\text{int})^0\}
\]

In this case it is clear that the safety requirements are manifested by the type explicitly given to the process thunk that the channel \(c\) expects to receive and to run. The interface of the incoming process should consist at most of channels \(a\) and \(b\) of type \((\text{int})^I\) resp. \((\text{int})^0\). The encoding of this process in the \(\lambda\pi_{\alpha\beta}\) calculus may be rendered as

\[
c ? (X : \langle \text{proc} \rangle)(\text{run } X)[\{a^-, b^+\}]
\]

Suppose that \(c\) is of type \((x : (\text{int})^I) \rightarrow [\Delta_{ab}]\), and the process in question is

\[
c ? (X : ((x : (\text{int})^I) \rightarrow [\Delta_{ab}])(Xa)
\]

What we now have is a channel \(c\) that admits incoming agents of type \(\Delta_{ab}\), which is similar to \(\Delta_{ab}\) except that the channel \(a\) in this type is abstracted. What is the safety requirement expressed by this type? We could say that the requirement is that the incoming process must, on being applied to any channel \(a\), result in a process whose interface is a couple of channels \(a\) and \(b\) of a given type. This could be encoded by

\[
c ? (X : ((\text{int} \rightarrow \text{proc}))(Xa)[\{a^-, b^+\})
\]
Consider now the following process:

\[ c?(X : \tau_x, y : (\text{int})^1)Xy \]

where

\[ \tau_x \overset{\text{def}}{=} (x : (\text{int})^1) \rightarrow [\Delta_{x^+}] \]

In this case the channel instantiating \( x \) in \( \Delta_{x^+} \) is sent along \( c \) together with an agent of type \( \tau_x \). We can now encode the process in question as

\[ c?(X : (\text{int} \rightarrow \text{proc}), y : (\text{int}))(Xay)[\{a^-, y^+\}] \]

If we instead have a process of form

\[ c?(X : \tau_{xy}, x : (\text{int})^1, y : (\text{int})^0)Xxy \]

where

\[ \tau_{xy} \overset{\text{def}}{=} (x : (\text{int})^1) \rightarrow (y : (\text{int})^0) \rightarrow [\Delta_{xy}] \]

then the encoding should be

\[ c?(X : (\text{int} \rightarrow \text{int} \rightarrow \text{proc}), x : (\text{int}), y : (\text{int}))(Xxy)[\{x^-, y^+\}] \]

But this scheme is not general enough. Consider the process

\[ c?(X : \tau_x, Z : (\tau_x \rightarrow \rho))ZX \]

where \( \rho \) is any suitable type. In this case, \( X \) is given as argument to an unknown agent \( Z \). In order to encode the agent \( X \) in a way that preserves the desired behaviour in all contexts, we may wrap \( X \) within an abstraction of form

\[ \lambda(y)((Xy)[\{y^-, b^+\}]) \]

Hence, no matter how \( X \) is used by the agent instantiating \( Z \), the interface of \( X \), whenever \( X \) is applied to a channel of the required type, will be of the desired form, since the filtered channel \( y^+ \) is bound by the occurrence of \( y \) in \( \lambda(y) \). This technique may be generalised to any number and type of arguments. For instance, consider a process of form

\[ c?(X : \tau_{xy}, Y : (\tau_x \rightarrow \rho))YX \]

where

\[ \tau_{xy} \overset{\text{def}}{=} (x : (\text{int})^1) \rightarrow (Z : \sigma_H) \rightarrow (y : (\text{int})^0) \rightarrow [\Delta] \]

for suitable types \( \sigma_H, \rho \), and \([\Delta]\). As a first approximation this process may be encoded, leaving aside type annotations, as

\[ c?(X, Y)Y(\lambda(x)\lambda(Z)\lambda(y)(Xxy)F) \]

where

\[ F \overset{\text{def}}{=} \mathcal{I}_c([\Delta]) \]

The definition of \( \mathcal{I}_c \) is given in the next section. The set \( \mathcal{I}_c([\Delta]) \) should include the set \( \{x^-, y^+\} \) if channels \( x \) and \( y \) occur in the domain of \( \Delta \). The set \( \mathcal{I}_c([\Delta]) \) includes
the domain of \( \Delta \) extended with all channels in \( \Gamma \) that may be imported by the process that instantiates \( X \).

However, the encoding is not complete since \( W \) and \( Z \) should also be wrapped according to its type \( \sigma_H \). Assume

\[
\sigma_H = (z : (\text{int}^1)) \rightarrow [z : (\text{int}^1)]
\]
i.e. the type of \( Z \) is a process thunk which upon receiving a channel \( z \) results in a process with interface \([z : (\text{int}^1)]\). Assume also that

\[
G \overset{\text{def}}{=} I_t((\tau_x Z y \rightarrow \rho))
\]
and

\[
E \overset{\text{def}}{=} I_t([z : \text{int}^1])
\]

Then the process can be encoded as

\[
c? (X, Y) (\lambda (W) (Y W) [G] (\lambda (x) \lambda (Z) (\lambda (y) (X x ((\lambda z) (Z z)) [E] y)) [F])
\]

The formalisation of these ideas are given in the next section.

### 4.8.3 Encoding \( \lambda \pi_v \) into \( \lambda \pi_v \beta \)

The formalisation of the wrapping of a higher order variable \( X \) in an environment \( \Gamma \), briefly \( F_t(X) \), is given in Figure 4.14. The definition of \( I_t \) is given in Figure 4.15. It consists of the least set \( M \) of polarised channels including the set of channels occurring in the domain of \( \Gamma \), and closed with respect to any channels in \( \Gamma \) that may be imported along a channel occurring in \( M \) with negative polarity. The operator \( D(\Gamma, (\tau_1, \ldots, \tau_n)) \) collects all pairs \((u : \sigma)\) in \( \Gamma \) where \( u \) denotes a channel and \( \sigma \) is a subtype of one of the types \( \tau_1, \ldots, \tau_n \).

The predicate \( C_t([\Delta]) \) denotes the channels in \( \Gamma \), together with their types, that may possibly occur as the subject of an action in a process of type \( \Delta \). The predicate \( I_t([\Delta]) \) denotes the polarised channels occurring in the domain of \( C_t([\Delta]) \). If a channel \( a \) in \( C_t([\Delta]) \) has the capability to do an input action, then \( a^- \in I_t([\Delta]) \). Analogously, \( a^+ \in I_t([\Delta]) \) if \( a \) may eventually execute an output action.

The translation from \( \lambda \pi_v \) to \( \lambda \pi_v \beta \), \( F_t \), is given in Figure 4.16. This translation extends to abstractions and types. Basically, \( F_t \) is a homomorphism over all process operators, types and abstractions. The interesting point is the translation of a process variable, which is done according to its type, and is defined by
$C_r([[]]) = \emptyset$
$C_r(proc) = D(\Gamma, T)$
$C_r([\Delta, x : \sigma]) = C_r(\Delta), C_r(x : \sigma)$
$C_r(\sigma_H) = \emptyset$
$C_r(\sigma_H \rightarrow \rho) = C_r(\rho)$
$C_r((x : \sigma) \rightarrow \rho) = C_r(x : \sigma) \rightarrow C_r(\rho)$
$C_r(x : T_r(\langle S_L, S_R \rangle))) = [x : \langle S_L, S_R \rangle], closure(S_L)$
$closure_r(T) = \emptyset$
$closure_r((\tau_1, \ldots, \tau_n)) = D(\Gamma, (\tau_1, \ldots, \tau_n)), C_r(\tau_1), \ldots, C_r(\tau_n)$

$T_r([\Delta]) = \text{interface}(C_r([\Delta]))$

$\text{interface}([[]]) = \emptyset$
$\text{interface}(\Delta, x : \langle S_L, S_R \rangle)) = \text{interface}(\Delta) \cup P_-(x, S_L) \cup P_+(x, S_R)$

$P_-(u, S) = \begin{cases} \{u^-\} & \text{if } S \neq \top \\ \emptyset & \text{else} \end{cases}$
$P_+(u, S) = \begin{cases} \{u^+\} & \text{if } S \neq \bot \\ \emptyset & \text{else} \end{cases}$

$D(\emptyset, \sigma) = \emptyset$
$D(\Gamma, \sigma_H) = \emptyset$
$D(\Gamma, (\tau_1, \ldots, \tau_n)) = D(\Gamma, \tau_1), \ldots, D(\Gamma, \tau_n)$
$D(\Gamma, x : \sigma_H), \sigma) = D(\Gamma, \sigma)$
$D(\Gamma, u : \sigma'), \sigma) = \begin{cases} \{(u : T_r(\sigma'))\} & \text{if } \sigma \leq \sigma' \\ D(\Gamma, \sigma) & \text{else} \end{cases}$

Figure 4.15: Closure and environment interface
However, we defined a notion of interface that is to recover this notion in a setting where no limitations are imposed on the kind of processes that may be done by establishing a relation between the interfaces of \( \Gamma \) and \( \tau \), and thus we shall sometimes drop the index and write \( \mathcal{T}(\Gamma') \) and \( \mathcal{T}(\tau) \) instead of \( \mathcal{T}(\Gamma') \) and \( \mathcal{T}(\tau) \).

### 4.8.4 Adequacy of Translation

In order to justify the translation we need some notion of adequacy between a process \( P \) and its translation \( \mathcal{T}_v(P) \). Simulation or other forms of behavioural equivalence are not enough here, since we do not desire \( \mathcal{T}_v(P) \) to be indistinguishable from \( P \), but instead to encode security requirements in a more liberal context. This may be done by establishing a relation between the interfaces of \( P \) and \( \mathcal{T}_v(P) \). However, we do not have yet a notion of interface for the \( \lambda \pi, \beta \) calculus. In order to recover this notion in a setting were no limitations are imposed on the kind of processes that may transmitted and received, we define a notion of interface that is not part of the type system of the \( \lambda \pi, \beta \) calculus, and that determines the channels along which a process may communicate with external processes, which may be limited by the filter operator. In this context, the most general type \( \text{proc} \) denotes the interface of a process which may communicate along any channel defined in.

\[
\mathcal{T}_v(X) \overset{\text{def}}{=} \mathcal{F}_v(X).
\]

The translation of a channel type \( \langle S_1, S_0 \rangle \) is defined as either the translation of \( S_0 \), if \( S_0 \neq \bot \), or otherwise the translation of \( S_1 \). All process types are reduced to the type \( \text{proc} \). Observe that for types and environments the transformation \( \mathcal{T}_v \) is independent of the index \( \Gamma \), and thus we shall sometimes drop the index and write \( \mathcal{T}(\Gamma') \) and \( \mathcal{T}(\tau) \) instead of \( \mathcal{T}_v(\Gamma') \) and \( \mathcal{T}_v(\tau) \).
\[
\begin{align*}
S_r(\sigma_B) &= \sigma_B \\
S_r(\sigma_H \rightarrow \rho) &= T_r(\sigma_H) \rightarrow S_r(\rho) \\
S_r((x: \sigma) \rightarrow \rho) &= (x : T_r(\sigma)) \rightarrow S_r(x : \sigma)(\rho) \\
S_r((S_1, S_0)) &= T_r((S_1, S_0)) \\
S_r((\tau_1, \ldots, \tau_n)) &= T_r((\tau_1, \ldots, \tau_n)) \\
S_r(\text{proc}) &= \text{proc} \\
S_r([\Delta]) &= C([\Delta])
\end{align*}
\]

Figure 4.17: Interface reduction

<table>
<thead>
<tr>
<th>Type</th>
<th>α, β, γ ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>ρ := π</td>
</tr>
<tr>
<td>Base</td>
<td>σ_B := \text{unit}</td>
</tr>
<tr>
<td>Process</td>
<td>π := proc</td>
</tr>
<tr>
<td>IProc</td>
<td>π := π</td>
</tr>
<tr>
<td>HOValue</td>
<td>σ_H := σ_B</td>
</tr>
<tr>
<td>Channel</td>
<td>σ := {S}</td>
</tr>
<tr>
<td>Value</td>
<td>τ := σ_H</td>
</tr>
<tr>
<td>Sort</td>
<td>S := (\tau_1, \ldots, \tau_n)</td>
</tr>
<tr>
<td>HO-IValue</td>
<td>ς := π</td>
</tr>
</tbody>
</table>

| Environment | Channel : Δ ::= \emptyset | Δ, u : σ |
|             | Environment : Γ ::= \emptyset | Γ, x : τ | Γ, α : σ |

| Abbreviation | thunk type : ⟨Δ⟩ \text{ def} := \text{unit} \rightarrow [\Delta] |

Figure 4.18: Interface Types for the λπvβ calculus

The new notion of type is formalised in Figure 4.17, where a translation from types in \(\lambda\pi_v\) to interface types in \(\lambda\pi_v\beta\) is given.

The translation of \((x : \sigma) \rightarrow \rho\) preserves the reference to channel \(x\), whereas the type \(σ_H \rightarrow \rho\) is reduced to \(T_r(σ_H) \rightarrow S_r(ρ)\), since no binding in \(σ_H\) affects the type that results from an application of an abstraction of type \(σ_H \rightarrow ρ\) to a value of type \(σ_H\). The important point to observe here is that a process type \([Δ]\) is reduced to \(C([Δ])\), the closure of \([Δ]\) in the environment \(Γ\). The intention here is that a process \(P\) of this type must filter all possible channels in the environment \(Γ\) that may be imported via input. We decided to limit possible communications only along channels in \(Γ\), as explained above, because we believe that this better reflects the intention behind process types in the \(\lambda\pi_v\) calculus. We think it is fair to assume that a process will only allow communication along the channels it knows, as is the case e.g. in Java applets. However, unknown channels may also become part of the interface by input and binding to a filter operator.

The definition of the interface types for the \(\lambda\pi_v\beta\) calculus is given in Figure 4.18. If a process or abstraction has interface type \(ς\), we write \(P :: ς\). Interface process types are ranged over by \(ς\), and other HO interface types by \(ς\).

We also have to reintroduce subtyping. The new definition of well-formed types is given in Figure 4.19, and is straightforward.
The resulting type system for interface types is given in Figure 4.20. It is only a slight variation over the type system for the $\lambda\pi_v$ calculus. The most important novelty is the rule IFILT, which tells that if $P :: \omega$ for some type $\omega$, then $P[L]$, i.e. filtering the interface of $P$ by the polarised channels in the set $L$, results in a process whose interface is the restriction of the interface $\omega$ to the channels occurring in $L$. Other rules are analogous to the rules given for the type system of the $\lambda\pi_v$ calculus.

We get the following subject reduction result, whose proof is similar to the proof of the corresponding theorem for the $\lambda\pi_v$ calculus. In this case, a corresponding assertion for labelled transitions must also be proved, and special care must be taken with regard to bound input.

**Theorem 3: Subject Reduction for Interface Types**

If $\Gamma \vdash P :: \omega$ and $P \rightarrow P'$, then $\Gamma \vdash P' :: \omega$

We may establish the following type safety theorem. The type safety rules are given in Figure 4.21.

**Theorem 4: Type safety for Interface Types**

If $\Gamma \vdash P :: \omega$ then $P \nvdash_{\text{err}}$

**Proof** By induction on the derivation of $\Gamma \vdash P :: \omega$.

We now formulate a pair of theorems showing that the translation is adequate. The first one says that if a process $P$ has an interface that is described by the type $\pi$, the translation $\mathcal{T}_\pi(P)$ does not introduce new capabilities, which guarantees that a protection policy upon resource access is not changed in the translation.
(Common)

(IVAL) \( \Gamma, u : \tau, \Gamma' \vdash \text{Env} \)
(ICON) \( \Gamma \vdash \text{Env} \)

(ISUBH) \( \Gamma \vdash P :: \varsigma \quad \Gamma \vdash \varsigma \leq \varsigma' \)

(Function)

(IABS_H) \( \Gamma, X : \sigma_{B} \vdash P :: \varsigma \quad \Gamma \vdash \lambda(X : \sigma_{B})P :: \sigma_{B} \rightarrow \varsigma \)

(IAPP_H) \( \Gamma \vdash P :: \sigma_{B} \rightarrow \varsigma \quad \Gamma \vdash Q :: \sigma_{B} \)
\( \Gamma \vdash PQ :: \varsigma \)

(IABS_N) \( \Gamma, x : \sigma \vdash P :: \varsigma \quad \Gamma \vdash \lambda(x : \sigma)P :: (x : \sigma) \rightarrow \varsigma \)

(IAPP_N) \( \Gamma \vdash P :: (x : \sigma) \rightarrow \varsigma \quad \Gamma \vdash u : \sigma \)
\( \Gamma \vdash Pu :: \varsigma\{u/x\} \)

(Process)

(INL) \( \Gamma \vdash \text{Env} \quad (\text{IPAR}) \quad \Gamma \vdash P_{1} :: \varnothing \quad \Gamma \vdash P_{3} :: \varnothing \)

(IOUT) \( \Gamma \vdash u : \{(\tau_{1}, \ldots, \tau_{n})\} \quad \Gamma \vdash P :: \varnothing \quad \Gamma \vdash V_{1} : \tau_{1} \quad \Gamma \vdash V_{i} : \tau_{i} \quad \Gamma \vdash u!(V_{1}, \ldots, V_{n})P :: \varnothing \)

(IIN) \( \Gamma \vdash u : \{(\tau_{1}, \ldots, \tau_{n})\} \quad \Gamma \vdash x_{1} : \tau_{1}, \ldots, x_{n} : \tau_{n} \quad \Gamma \vdash P :: \varnothing \quad x_{1} : \tau_{1}, \ldots, x_{n} : \tau_{n} \)
\( \Gamma \vdash u?(x_{1}, \ldots, x_{n})P :: \varnothing \)

(IFILT) \( \Gamma \vdash P :: \varnothing \quad \Gamma \vdash P|L :: \varnothing \uparrow L \)

\( \varnothing \vdash \Gamma \vdash a : \sigma \quad \text{def} \quad \Gamma \vdash u : \sigma \leq \varnothing \)

\( \varnothing, x : \sigma \quad \text{def} \quad [\varnothing, x : \sigma] \)

\( \varnothing, x : \sigma \quad \text{def} \quad \varnothing \)

\( \varnothing \uparrow L \quad \text{def} \quad [u : \sigma \in \varnothing \mid u \in L] \)

If \( \varnothing = \text{proc in } \Gamma \) above, then \( \varnothing = [\Gamma'] \) where \( \Gamma' \) is \( \Gamma \) restricted to the domain of channel names and channel variables.

Figure 4.20: Interface typing system for \( \lambda \pi \)

\[ \frac{a'(x_{1} : \tau_{1}, \ldots, x_{n} : \tau_{n})P \overset{\tau_{n}}{\longrightarrow}_{evr}}{\varnothing \vdash [a : (\langle x_{1} : \tau_{1}, \ldots, x_{n} : \tau_{n} \rangle)] \leq \varnothing} \]

\[ \frac{a!(\langle V_{1}, \ldots, V_{n} \rangle)}{P \overset{\tau_{n}}{\longrightarrow}_{evr} \text{ if } \text{not } \tau_{i} \text{ s.t. } \Gamma \vdash [a : \langle t_{1}, \ldots, \tau_{n} \rangle] \leq \varnothing} \]

\[ \frac{P \overset{\tau_{n}}{\longrightarrow}_{evr} \text{ or } Q \overset{\tau_{n}}{\longrightarrow}_{evr} \text{ if } P \overset{\tau_{n}}{\longrightarrow}_{evr} \text{ and } Q \overset{\tau_{n}}{\longrightarrow}_{evr}}{P \overset{\tau_{n}}{\longrightarrow}_{evr} Q \overset{\tau_{n}}{\longrightarrow}_{evr} \text{ or } \tau_{n} \overset{\tau_{n}}{\longrightarrow}_{evr}} \]

Figure 4.21: Typing safety for interface types
Nevertheless, the type of a process in the $\lambda\pi_r$ calculus does not give a complete characterisation of the channels along which the process may communicate with the environment, mainly because channels may be imported and then used for communication. In consequence, the domain of $S_r(\pi)$ in Theorem 5 must include channels that may become part of the interface by being imported during execution.

This first result is not sufficient, since by subtyping it is valid if we let all process be mapped onto 0. We must thus show that the interface given by $S_r(\pi)$ is in some well-defined sense not less than that of $\pi$, i.e. that the translation $T_r(P)$ preserves the interface of $P$. This may be formalised by stating that if $P$ may be given type $\pi$, and $T_r(P)$ type $\pi'$, then the restriction of the domain of $\pi$ to those channels that appear in the domain of $\pi'$ is still a valid type for $P$. This means that any type $\pi'$ for $T_r(P)$ will always include the channels along which $P$ may communicate with its environment.

**Theorem 5** If $\Gamma \vdash P : \pi$, then $S_r(\Gamma) \vdash T_r(P) :: S_r(\pi)$.

**Proof** We prove the following more general proposition:

If $\Gamma \vdash P : \rho$ then $T_r(\Gamma) \vdash T_r(P) :: S_r(\rho)$

The proof is by induction on derivation of $\Gamma \vdash P : \rho$, and uses the fact that the typing rules are syntax directed and deterministic, except for the subtyping rules. We show a few cases:

**(IN)** Assume $\Gamma \vdash u?(x_1 : \tau_1, \ldots, x_n : \tau_n)P : \pi$.

Then by (IN)

$$\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash P : \pi, x_1 : \tau_1, \ldots, x_n : \tau_n \text{ and } \pi \vdash_r u : (\tau_1, \ldots, \tau_n)$$

By induction

$$T(\Gamma), x_1 : T(\tau_1), \ldots, x_n : T(\tau_n) \vdash T_r(P) :: S_r(\pi, x_1 : \tau_1, \ldots, x_n : \tau_n)$$

where $\Gamma' = \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n$.

Now

$$(1) \quad S_r(\pi) \vdash T_r(\Gamma') \ u : (T(\tau_1), \ldots, T(\tau_n))$$

follows from $\pi \vdash_r u : (\tau_1, \ldots, \tau_n)$, the definitions of $S_r$ and $T_r$, and the fact, which may be proved easily, that if $\Gamma \vdash \tau_1 \leq \tau_2$ then $T_r(\Gamma) \vdash S_r(\tau_1) \leq S_r(\tau_2)$.

By the definition of $\pi \vdash_r u : (\tau_1, \ldots, \tau_n)$ we deduce that

$$\mathcal{C}(\pi, x_1 : \tau_1, \ldots, x_n : \tau_n) = \mathcal{C}(\pi), x_1 : \tau_1, \ldots, x_n : \tau_n$$

and hence

$$S_r(\pi, x_1 : \tau_1, \ldots, x_n : \tau_n) = T(\mathcal{C}(\pi, x_1 : \tau_1, \ldots, x_n : \tau_n)) = T(\mathcal{C}(\pi), x_1 : T(\tau_1), \ldots, x_n : T(\tau_n)) = S_r(\pi, x_1 : T(\tau_1), \ldots, x_n : T(\tau_n)).$$

From this and (1), by (IN) we get

$$\Gamma \vdash u? (x_1 : T(\tau_1), \ldots, x_n : T(\tau_n)) : T_r(P) :: S_r(\pi)$$

as desired.

**(APP)** Assume $\Gamma \vdash Pu : \rho \{u/x\}$. Then, by (APP)
\[ \Gamma \vdash P : (x : \sigma) \rightarrow \rho, \quad \Gamma \vdash u : \sigma \]

By induction

\[ \mathcal{T}(\Gamma) \vdash \mathcal{T}_r(P) :: \mathcal{S}_r((x : \sigma) \rightarrow \rho) \equiv (x : \mathcal{T}_r(\sigma)) \rightarrow \mathcal{S}_r(\rho) \] and \( \mathcal{T}(\Gamma) \vdash u : \mathcal{T}(\sigma) \)

Then, by (IAPP) \( \mathcal{T}(\Gamma) \vdash \mathcal{T}_r(P)u :: \mathcal{S}_r(\rho)\{u/x\} \)

Now, by the definitions of \( \mathcal{T}_r \) and \( \mathcal{S}_r \) it is easy to show that

\[ \mathcal{T}_r(P)u = \mathcal{T}_r(Pu) \text{ and } \mathcal{S}_r(\rho)\{u/x\} = \mathcal{S}_r(\rho\{u/x\}) \]

Hence

\[ \mathcal{T}_r(\Gamma) \vdash \mathcal{T}_r(Pu) :: \mathcal{S}_r(\rho\{u/x\}) \]

as desired.

(\textsc{ABS}_N) Assume \( \Gamma \vdash \lambda(x : \sigma)P : (x : \sigma) \rightarrow \rho \). Then, by (\textsc{ABS}_N)

\[ \Gamma, x : \sigma \vdash P : \rho \]

By induction

\[ \mathcal{T}_r(\Gamma), x : \mathcal{T}(\sigma) \vdash \mathcal{T}_r, x : \sigma(P) :: \mathcal{S}_{\mathcal{T}, x : \sigma}(\rho) \]

Then, by (I\textsc{ABS}_N)

\[ \mathcal{T}_r(\Gamma) \vdash \lambda(x : \mathcal{T}(\sigma))\mathcal{T}_r, x : \sigma(P) :: (x : \mathcal{T}(\sigma)) \rightarrow \mathcal{S}_{\mathcal{T}, x : \sigma}(\rho) \]

Now

\[ x : \mathcal{T}(\sigma) \equiv \mathcal{T}(x : \sigma) \text{ and } \mathcal{T}(x : \sigma) \rightarrow \mathcal{S}_{\mathcal{T}, x : \sigma}(\rho) \equiv \mathcal{S}_r((x : \sigma) \rightarrow \rho) \]

Also

\[ \lambda(x : \mathcal{T}(\sigma))\mathcal{T}_r, x : \sigma(P) \equiv \mathcal{T}_r(\lambda(x : \sigma)P) \]

Hence,

\[ \mathcal{T}(\Gamma) \vdash \mathcal{T}_r(\lambda(x : \sigma)P) :: \mathcal{S}_r((x : \sigma) \rightarrow \rho) \]

as desired.

(\textsc{VAL}) This is the most interesting case. We have

\[ \Gamma, u : \tau, \Gamma' \vdash u : \tau \]

We cannot use induction here, since what we have to prove is that

\[ \mathcal{T}(\Gamma), u : \mathcal{T}(\tau), \mathcal{T}(\Gamma') \vdash \mathcal{T}_{\mathcal{T}, u}(u) :: \mathcal{S}_{\mathcal{T}, u}(\tau) \]

where \( \Gamma'' = \Gamma, u : \tau, \Gamma' \).

94
We show this directly instead. We prove a more general statement. For any expression \( E \) and environment \( \Gamma \) such that \( \Gamma \vdash E : \tau \), we show that

\[
\mathcal{T}(\Gamma) \vdash \mathcal{F}_{\tau}(E, \tau) :: \mathcal{S}_{\tau}(\tau)
\]

The result will follow since \( \mathcal{T}_{\nu}(u) = \mathcal{F}_{\nu}(u, \tau) \). The proof is by induction on \( \tau \)'s formation.

If \( \tau = \sigma_H \) or \( \tau = \sigma \), the result follows by (ICON) since in this case \( E \) is a channel, channel variable, or literal, and thus \( \mathcal{T}(E) = E \) and \( \mathcal{S}_{\tau}(\tau) = \tau \).

If \( \tau = (y : \sigma) \to \rho \) we have two possibilities, according to whether \( \rho = \pi \) for some process type \( \pi \), or \( \rho = \sigma_H \) for some HO type \( \sigma_H \).

Assume \( \rho = \sigma_H \). Then we have to show that

\[
\mathcal{T}(\Gamma) \vdash \mathcal{F}_{\tau}(E, (y : \sigma) \to \sigma_H) :: \mathcal{S}_{\tau}(\tau)
\]

In other words, we have to show that

\[
\mathcal{T}(\Gamma) \vdash \lambda y : \tau(\mathcal{F}_{\tau,y,\sigma}(Ey, \sigma_H) :: (y : \mathcal{T}(\sigma)) \to \mathcal{S}_{\tau}(\sigma_H))
\]

By (IABS), it is enough to show that

\[
\mathcal{T}(\Gamma), y : \mathcal{T}(\sigma) \vdash \mathcal{F}_{\tau,y,\sigma}(Ey, \sigma_H) :: \mathcal{S}_{\tau}(\sigma_H)
\]

But this follows by induction.

Assume now that \( \rho = \pi \). We need to show in this case that

\[
\mathcal{T}(\Gamma), y : \mathcal{T}(\sigma) \vdash \mathcal{F}_{\tau,y,\sigma}(Ey, \pi) :: \mathcal{S}_{\tau}(\pi)
\]

i.e.

\[
\mathcal{T}(\Gamma), y : \mathcal{T}(\sigma) \vdash E[y]([I_{\tau,y,\sigma}(\pi)) :: \mathcal{C}_{\tau}(\pi)]
\]

By (IFILT), this is true if

\[
(a) \quad \mathcal{T}(\Gamma), y : \mathcal{T}(\sigma) \vdash Ey :: \mathcal{proc}
\]

and

\[
(b) \quad \mathcal{C}_{\tau}(\pi) = \mathcal{proc} \uparrow I_{\tau}(\pi) \text{ for } \mathcal{proc} \text{ in } \Gamma
\]

Assertion (b) follows easily from the definition of closure, and (a) from the fact that \( \mathcal{proc} \) is the most general process type, hence the result follows by subtyping.

The case \( \tau = \sigma_H \to \rho \) is similar.

**End of Proof of Theorem 5**

In order to formulate the next result we need the following definition:

**Definition: Interface Restriction:** The restriction of the domain of the type \( \pi \) of a process in the \( \lambda \pi \_ \) calculus, to the channels appearing in the type \( \varpi \), briefly \( \pi \_ \varpi \), is defined as follows:

\[
\pi \_ \varpi = \{ (u : \tau) \in \pi \text{ s.t. } u \in \text{domain}(\varpi) \}
\]
The definition of restriction is extended to arbitrary but structurally equivalent types as follows:

\[
\begin{align*}
(\sigma_H \rightarrow \rho)[(\sigma'_H \rightarrow \varsigma) &= \sigma_H \rightarrow (\rho[\varsigma]) \\
((x : \sigma) \rightarrow \rho)[(x : \sigma') \rightarrow \varsigma) &= (x : \sigma') \rightarrow (\rho[\varsigma])
\end{align*}
\]

We may now state the following result:

**Theorem 6** If \(\Gamma \vdash P : \pi\), and \(\mathcal{T}(\Gamma) \vdash \mathcal{T}_r(P) :: \varpi\), then \(\Gamma \vdash P : \pi[\varpi]\)

**Proof:** We prove the following more general proposition:

If \(\Gamma \vdash P : \rho\), and \(\mathcal{T}(\Gamma) \vdash \mathcal{T}_r(P) :: \varsigma\), then \(\Gamma \vdash P : \rho[\varsigma]\).

The proof is by induction on the derivation of \(\mathcal{T}_r(\Gamma) \vdash \mathcal{T}_r(P) :: \varsigma\), assuming that the last step of derivation of \(\Gamma \vdash P : \rho\) is not \((\text{SUB}_H)\). No generality is lost here, since a rule other than \((\text{SUB}_H)\) must always have been applied before one or more successive occurrences of \((\text{SUB}_H)\) in a deduction tree, and this rule is uniquely determined by the syntax of \(P\), consequently by the syntax of \(\mathcal{T}_r(P)\) since \(\mathcal{T}_r\) is a homomorphism, except when \(P\) is a process variable, whose case is treated separately below. The result will then follow since if \(\rho \leq \rho'\) then \(\rho[\varsigma] \leq \rho'[\varsigma]\), and the result follows, possibly after similar applications of the rule \((\text{ISUB}_N)\).

We show a few cases:

**(IIN)** Assume

\(\Gamma \vdash u ?(x_1 : \tau_1, \ldots, x_n : \tau_n).P :: \pi\)

and

\(\mathcal{T}(\Gamma) \vdash u ?(x_1 : \mathcal{T}(\tau_1), \ldots, x_n : \mathcal{T}(\tau_n)).\mathcal{T}_r(P) :: \varpi\)

By (IIN)

\(\mathcal{T}(\Gamma), x_1 : \mathcal{T}(\tau_1), \ldots, x_n : \mathcal{T}(\tau_n) \vdash \mathcal{T}_r(P) :: \varpi, x_1 : \mathcal{T}(\tau_1), \ldots, x_n : \mathcal{T}(\tau_n)\)

and

\(\varpi \vdash \mathcal{T}(\Gamma) u :: (\mathcal{T}(\tau_1), \ldots, \mathcal{T}(\tau_n))\)

Hence \(u\) belongs to the domain of \(\varpi\). By induction

\(\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \mathcal{P} : (\pi, x_1 : \tau_1, \ldots, x_n : \tau_n)[(\varpi, x_1 : \mathcal{T}(\tau_1), \ldots, x_n : \mathcal{T}(\tau_n))\]

Now the following holds

\((\pi, x_1 : \tau_1, \ldots, x_n : \tau_n)[(\varpi, x_1 : \mathcal{T}(\tau_1), \ldots, x_n : \mathcal{T}(\tau_n)) \equiv (\pi[\varpi], x_1 : \tau_1, \ldots, x_n : \tau_n)\)

Also, since \(u\) belongs to the domain of \(\varpi\) and by (IN)

\(\pi \vdash_r u : (\tau_1, \ldots, \tau_n)^I\)

we obtain

\(\pi[\varpi] \vdash_r u : (\tau_1, \ldots, \tau_n)^I\)
Hence, by (IN)

$$\Gamma \vdash (\pi[\varnothing], x_1 : \tau_1, \ldots, x_n : \tau_n)$$

Since

$$(\pi[\varnothing], x_1 : \tau_1, \ldots, x_n : \tau_n) = (\pi, x_1 : \tau_1, \ldots, x_n : \tau_n)[(\varnothing, x_1 : \tau_1, \ldots, x_n : \tau_n)$$

the result follows.

(IAAP_N) Assume

$$\Gamma \vdash Pu : \rho\{u/x\} \text{ and } T(\Gamma) \vdash T_r(P)u : \varsigma\{u/x\}$$

By (APP_N)

$$\Gamma \vdash P : (x : \sigma) \rightarrow \rho, \ \Gamma \vdash U : \sigma$$

and by (IAPP_N)

$$T(\Gamma) \vdash T_r(P) :: (x : \sigma') \rightarrow \varsigma \text{ and } T(\Gamma) \vdash U : \sigma'$$

By induction

$$\Gamma \vdash P : ((x : \sigma) \rightarrow \rho)[(x : \sigma') \rightarrow \varsigma] \equiv (x : \sigma) \rightarrow (\rho[\varsigma])$$

Hence, by (APP_N)

$$\Gamma \vdash Pu : (\rho[\varsigma]\{u/x\}) \equiv (\rho\{u/x\})[(\varsigma\{u/x\})]$$

as desired.

(IABS_H) Assume

$$\Gamma \vdash \lambda(X : \sigma_H)P : \sigma_H \rightarrow \rho$$

and

$$T(\Gamma) \vdash \lambda(X : T(\sigma_H))T_r(P) :: T(\sigma_H) \rightarrow \varsigma$$

Then, by (ABS_H) and (IABS_H)

$$\Gamma, X : \sigma_H \vdash P : \rho \text{ and } T(\Gamma), X : T(\sigma_H) \vdash T_r(P) :: \varsigma$$

By induction

$$\Gamma, X : \sigma_H \vdash P : \rho[\varsigma]$$

Hence, by (ABS_H)

$$\Gamma \vdash \lambda(X : \sigma_H)P : \sigma_H \rightarrow \rho[\varsigma] \equiv (\sigma_H \rightarrow \rho)[(\sigma_H \rightarrow \varsigma)]$$

as desired.

(IVAL) The only case where the induction step in this proof cannot be applied is when we have

$$\Gamma, u : \tau, \Gamma' \vdash u : \tau, \ T(\Gamma), u : T(\tau), T(\Gamma') \vdash T_r(u) :: \varsigma$$
where $\Gamma'' = \Gamma, u : \tau, \Gamma'$. Here, since $\mathcal{T}_r(u) = \mathcal{F}_r(u, \tau)$ it suffices to show that for any $\tau$ and $\Gamma$

$$\text{if } \mathcal{T}(\Gamma) \vdash \mathcal{T}_r(u, \tau) :: \tau' \text{ and } \Gamma \vdash u : \tau \text{ then } \mathcal{T}(\tau) \leq \tau'$$

The proof of this assertion is straightforward, and can be done by induction on $\tau$'s formation. It follows ultimately from the definition of $\mathcal{F}_r$ for process types, i.e.

$$\mathcal{F}_r(x, \pi) = x[\mathcal{I}_r(\pi)]$$
since if $x$ has type $\pi$ the domain of $x[\mathcal{I}_r(\pi)]$ includes the domain of $\mathcal{T}(\pi)$.

Now, if $\mathcal{T}(\tau) \leq \tau'$, then $\tau' \tau = \tau$. Hence

$$\Gamma, u : \tau, \Gamma' \vdash u : \tau[\tau']$$
as desired. 

**End of proof of Theorem 6**

To complete our argument concerning the adequacy of the translation, we should also show some kind of behaviour equivalence relation between $P$ and $\mathcal{T}_r(P)$. It is clear that these processes should not be observational equivalent, since we want $\mathcal{T}_r(P)$ to have in some sense a more liberal behaviour than $P$. Nevertheless, it would be desirable that $\mathcal{T}_r(P)$ should at least be able to simulate $P$’s behaviour, i.e. that $P \subseteq \mathcal{T}_r(P)$ for a definition of a simulation relation $\subseteq$ based on the transition $\rightarrow$ and commitment. We will only sketch a proof of this claim. It is based on the fact that if

$$\Gamma \vdash P : \rho$$
then

$$P \subseteq \mathcal{F}_r, X : \rho(X)[\mathcal{T}_r(P)/X]$$
and more generally, for any agent expression $Q$ and environment $\Gamma$ such that $\Gamma \vdash Q : \rho$

$$Q[P_1/X_1, \ldots, P_n/X_n] \subseteq \mathcal{T}_r(Q)[\mathcal{T}_r(P_1)/X_1, \ldots, \mathcal{T}_r(P_n)/X_n]$$
where we assume that $P_1, \ldots, P_n$ have types such that

$$\Gamma \vdash Q[P_1/X_1, \ldots, P_n/X_n] : \rho$$
and that $Q[P_1/X_1, \ldots, P_n/X_n]$ is a closed agent. We claim that the relation $S$ on composed of the (closed) process pairs

$$(Q[P_1/X_1, \ldots, P_n/X_n], \mathcal{T}_r(Q)[\mathcal{T}_r(P_1)/X_1, \ldots, \mathcal{T}_r(P_n)/X_n])$$
such that

$$\Gamma \vdash Q : \rho \text{ and } \Gamma \vdash Q[P_1/X_1, \ldots, P_n/X_n] : \rho$$
is a simulation in the sense that if

$$(P, Q) \in S \text{ and } P \rightarrow P'$$
then for some $Q'$ such that $Q \rightarrow Q'$ we have $(P', Q') \in S$. The proof, which omitted here, is by induction on the the derivation of $Q \rightarrow Q'$.
Chapter 5

CRCHAM - The Context Reflexive CHAM

5.1 Introduction

We introduce here the Context Reflexive Chemical Abstract Machine, shortly CRCHAM, a notation based on the reflexive chemical abstract machine and the Join Calculus [47]. We extend the RCHAM with primitives supporting the notion of context. Informally, contexts are domains or areas of computation with an administrative policy concerning the movement of messages across the domain boundary. Certain messages, according to the name in subject position, are blocked, i.e. not allowed to either enter a domain to leave it, or both. This is enforced by the blocking operator, a set of names with polarities. As an illustration, a process of type $P \uparrow x^+$ does not allow any messages of type $x \langle \bar{v} \rangle$ to move inside the scope of the positive blocking operator, i.e. the transition

$$(P \uparrow x^+) \mid x \langle \bar{v} \rangle \rightarrow (P \mid x \langle \bar{v} \rangle) \downarrow x^+$$

is not allowed. By contrast, for any $y$ such that $y \neq x$,

$$(P \uparrow x^+) \mid y \langle \bar{v} \rangle \rightarrow (P \mid y \langle \bar{v} \rangle) \downarrow x^+$$

It is also possible for the message $y \langle \bar{v} \rangle$ to leave the domain of the positive blocking operator. This may be expressed by the structural congruence relation $\equiv$. Thus

$$(P \uparrow x^+) \mid y \langle \bar{v} \rangle \equiv (P \mid y \langle \bar{v} \rangle) \downarrow x^+$$

The negative blocking operator is intended to block certain messages from leaving a determined domain. The structural rule for this operator is

$$P \downarrow x^- \mid y \langle \bar{v} \rangle \equiv (P \mid y \langle \bar{v} \rangle) \uparrow x^-$$

on condition that $x \neq y$. Otherwise, the only applicable rule is

$$(P \mid x \langle \bar{v} \rangle \downarrow x^-) \rightarrow (P \downarrow x^-) \mid x \langle \bar{v} \rangle$$

We can also introduce an operator for blocking a name on both polarities at the same time. Such operator is not expressible in terms of negative and positive blocking, as is the case in the $\pi$-calculus with polarised blocking (see Chapter 4). This is an
important difference. The reason for this is that in the CRCHAM the crossing of a domain boundary is not always reversible, as illustrated by the transition rules above. Thus, an agent of type

\[ P \setminus x^- \setminus x^+ \]

would allow a message \( x(\overline{v}) \) to cross the outer boundary, but not the inner one. Once it has crossed the outer barrier it will be caught between both barriers and impeded to move in any direction:

\[ (P \setminus x^- \setminus x^+) \, | \, x(\overline{v})) \rightarrow ((P \setminus x^-) \, | \, x(\overline{v})) \setminus x^+ \]

In this context, the message \( x(\overline{v}) \) may be viewed as “sterilized”, and will never be able to react. Thus, we get the following equivalence:

\[ ((P \setminus x^-) \, | \, x(\overline{v})) \setminus x^+ \equiv P \setminus x^- \setminus x^+ \]

In order to express total blocking, i.e the conjunction of negative and positive blocking, in CRCHAM, we have two alternatives. The first one is to define an operator for total blocking with the rule

\[ P \setminus x \, | \, y(\overline{v}) \rightleftharpoons (P \setminus y(\overline{v})) \setminus x, \quad x \neq y \]

The other alternative is to allow agents blocked on a (finite) sets of names with polarities. We choose this second alternative for two reasons: it gives a simpler syntax and a more intuitive semantics, and we would like to extend the notation later with a kind of dual operator to blocking, the filtering operator, which can only be defined in terms of sets. Thus, the equivalent of total blocking on \( x \) is expressed by \( P \setminus \{x^+,x^-\} \).

Contexts may be nested in CRCHAM. The context structure in CRCHAM is hierarchical, and may be described as a tree. Reactions in CRCHAM are purely local. Each context may contain a set of reaction rules, which are active only within the context proper, but not within any subcontexts.

Besides the blocking operators, replication and the restriction operator are also introduced. In contrast to the RCHAM, in the CRCHAM we have several sets of reaction rules, one for each context. In order to simplify the semantics, we decided to incorporate reaction rules as part of the syntax, and introduce what we call reaction processes. A reduction in the CRCHAM may take place whenever a message pattern matching the join pattern of a reaction process and the reaction process itself occurs within the same context. The result is similar to the asynchronous version of the polyadic \( \pi \)-calculus. A reaction process of type \( x(\overline{v}) \rightarrow P \) may be viewed as a process \( P \) prefixed by the input prefix \( x(\overline{v}) \), i.e \( x(\overline{v}) \cdot P \). Similarly, a reaction process \( (x_1(\overline{v}_1) \, | \, x_2(\overline{v}_2)) \rightarrow P \) would correspond to

\[ x_1(\overline{v}_1).x_2(\overline{v}_2).P + x_2(\overline{v}_2).x_1(\overline{v}_1).P \]

However, in the CRCHAM a reaction is an atomic event. Furthermore, communication in the \( \pi \)-calculus with blocking is not local, and there is no correspondence in the \( \pi \)-calculus to transitions denoting message movement which are not reversible. It is largely due to these differences that we preferred to introduce the CRCHAM instead of working directly with an asynchronous version of the \( \pi \)-calculus with polarised blocking. In addition, reaction patterns are very convenient for modeling functions with state, which are suitable for modeling notions of state in object-oriented programming languages. Together with restriction and blocking, this gives
us very powerful tools for representing encapsulation and inheritance in an object-oriented setting.

The rest of this chapter is organised as follows. In Section 5.2 we develop the syntax and semantics of the CRCHAM. In Section 5.3 we show by examples how CRCHAM may be used to encode objects and common data structures. In Section 5.4 we present a higher-order version of the CRCHAM. Section 5.5 is dedicated to the presentation of an encoding of a fragment of Smalltalk-80. Finally, in Section 5.6 we sum up this chapter.

### 5.2 The Syntax and Semantics of the CRCHAM

The syntax of CRCHAM is the following:

\[
P \quad \text{def} \quad \text{processes}
\]

\[
0 \quad \text{null process}
\]

\[
x(\bar{v}) \quad \text{messages}
\]

\[
J \triangleright P \quad \text{reaction}
\]

\[
P \mid P \quad \text{parallel composition}
\]

\[
P \backslash L \quad \text{blocking}
\]

\[
(\nu x)P \quad \text{restriction}
\]

\[
[x = y]PQ \quad \text{if-then-else}
\]

\[
!P \quad \text{replication}
\]

\[
J \quad \text{def} \quad \text{join-patterns}
\]

\[
x(\bar{v}) \quad \text{message pattern}
\]

\[
J \mid J \quad \text{join of patterns}
\]

For notational convenience, we may write \{x, \ldots\} for \{x^+, x^-, \ldots\}. The definition of reaction (rtn), received (rcn) and free names (fn) are as follows

\[
\text{rcn}(x(\bar{v})) \quad \text{def} \quad \{u : u \in \bar{v}\}
\]

\[
\text{rcn}(J \mid J') \quad \text{def} \quad \text{rcn}(J) \cup \text{rcn}(J')
\]

\[
\text{rtn}(x(\bar{v})) \quad \text{def} \quad \{x\}
\]

\[
\text{rtn}(J \mid J') \quad \text{def} \quad \text{rtn}(J) \cup \text{rtn}(J')
\]

\[
\text{fn}(J \triangleright P) \quad \text{def} \quad \text{rtn}(J) \cup (\text{fn}(P) - \text{rcn}(J))
\]

\[
\text{fn}(x(\bar{v})) \quad \text{def} \quad \{x\} \cup \{u : u \in \bar{v}\}
\]

\[
\text{fn}(P \mid P') \quad \text{def} \quad \text{fn}(P) \cup \text{fn}(P')
\]

\[
\text{fn}([x = y]PQ) \quad \text{def} \quad \text{fn}(P) \cup \text{fn}(Q) \cup \{x, y\}
\]

\[
\text{fn}(P \backslash L) \quad \text{def} \quad \bar{L} \cup \text{fn}(P)
\]

\[
\text{fn}(\nu x)P \quad \text{def} \quad \text{fn}(P) - \{x\}
\]

\[
\text{fn}(!P) \quad \text{def} \quad \text{fn}(P)
\]

Here, \(\bar{L} = \{x : x^+ \in \overline{L} \lor x^- \in \overline{L}\}\). A name \(x\) is fresh in \(P\) if \(x \notin \text{fn}(P)\).
An active context is a process with a hole defined by the following syntax:

\[ C \overset{\text{def}}{=} P \mid \llbracket \cdot \rrbracket \mid C \mid C \setminus L \mid (\nu x)C \mid \lambda C \]

If \( C \) is any context \( C[P] \) is the result of filling the holes in \( C \) with the process \( P \).

The semantic rules are the following:

\[
\begin{align*}
\text{str-comp} & \quad P \mid Q \quad \Rightarrow \quad P, Q \\
\text{str-rep} & \quad \nu P \quad \Rightarrow \quad P \mid P \\
\text{str-match} & \quad [x = y]PQ \quad \Rightarrow \quad P \quad \text{if } x = y \\
\text{str-mismatch} & \quad [x = y]PQ \quad \Rightarrow \quad Q \quad \text{if } x \neq y \\
\text{str-rest1} & \quad (\nu x)P \mid Q \quad \Rightarrow \quad (\nu x')(P\{x'/x\} \mid Q) \quad x' \notin \text{fn}(P \mid Q) \setminus \{x\} \\
\text{str-rest2} & \quad ((\nu x)P) \setminus L \quad \Rightarrow \quad (\nu x')(P\{x'/x\} \setminus L) \quad x' \notin \text{fn}(P \setminus L) \setminus \{x\} \\
\text{red} & \quad J \triangleright P \mid J\sigma_{rcn} \quad \Rightarrow \quad P\sigma_{rcn} \\
\text{str-hont} & \quad (x(v), S) \setminus L \quad \Rightarrow \quad x(v), S \setminus L \quad \text{if } x^+ \notin L \\
\text{str-bin} & \quad x(v), S \setminus L \quad \Rightarrow \quad (x(v), S) \setminus L \quad \text{if } x^- \notin L \\
\text{str-ext} & \quad C[P] \quad \Rightarrow \quad C[P] \quad \text{if } P \rightarrow P'
\end{align*}
\]

The term \( J\sigma_{rcn} \) denotes the join pattern \( J \) with the names occurring in \( \text{ren}(J) \) substituted according to the substitution \( \sigma_{rcn} \), whereas \( P\sigma_{rcn} \) denotes the process \( P \) with the names occurring in \( \text{fn}(P) \) substituted according to \( \sigma_{rcn} \). In the latter case we assume that the substitution is hygienic, i.e. any bound names occurring in \( P \) are conveniently renamed to avoid capture.

The defined variables of a join pattern in the CRHAM have no counterpart in the CRCHAM. Hence, we refer to the names occurring in subject position in a join pattern in the CRCHAM as reaction names. In the CRCHAM, reaction rules are not associated with a single reaction site as in RCHAM, and join patterns do not statically bind the names occurring in it in subject position. Local definitions of type \textbf{def} \( D \) \textbf{in} \( P \) are thus not necessary. Furthermore, reaction processes are not implicitly replicated in the CRCHAM as is the case with reaction rules in the RCHAM. Hence, a reaction will only fire once, unless the corresponding reaction process is replicated. Reaction processes in the CRCHAM are nevertheless associated with the context in which it occurs, since they may not leave the context where they were created.

The components of a join pattern are polyadic terms of type \( x(y_1, \ldots, y_n) \), where \( n \geq 0 \), and the received names \( y_i \) are distinct and bound by their occurrence in the join pattern. The CRCHAM is untyped, hence no arity is associated with \( x \) in the expression above. As a result, \( x \) may occur in another join pattern or in a message with a different arity. It is also assumed that the set of received names of any two distinct components of a join pattern are disjoint. Also, the set of reaction names in a join pattern and its set of received names are disjoint. Nevertheless, reaction names may be repeated in a join pattern in the subject part of distinct components.

### 5.3 CRCHAM in Action

#### 5.3.1 Some introductory examples

An object may be encoded as a server accepting requests to execute any method defined in the interface of its class. In an object-oriented system it is usually assumed that each object has a unique identifier, which is used for all communication with the object. However, in an object-based open system an object might show towards its environment a more flexible interface, consisting of various names or aliases referring to the same object, and even of the names of services offered, which could
thus be accessed directly without the need to go through the object’s identifier name. Thus, an object asking for a certain service might content itself with getting it from whoever is offering the service in the environment. Moreover, groups of objects could also be accessed by letting these objects share a name, e.g. for use in multicast messages. Since we are going to concentrate upon object-oriented concepts, we will let each object have an identifier, but we do not assume that it is unique, a fact that cannot be guaranteed in open systems, or that services may not be directly offered.

**Cells**

We start with a simple example taken from [1], an object representing a simple cell. The class `cell` includes a field `contents`, which makes up the state of the cell. Its interface consists of two methods: `get`, that returns the value of `contents`, and `set`, that updates the value of `contents`, initially set to 0:

```plaintext
Class cell
  var contents := 0;
  method get() begin return self.contents end;
  method set(n) begin self.contents := n end;
end
```

We may define `cell` in CRCHAM as follows:

```
\cell(\) \triangleright ( \nu contents)(
  ( \nu contents(0) |
    ! (get(\) \triangleright contents(x) \bimap k(x) \triangleright contents(x)) |
    ! (set(\) \triangleright contents(x) \bimap contents(y)) ) ) \lor \{ get^+ , set^+ , contents \}
```

A state names, e.g. `contents` above, is blocked to avoid inconsistent states in case the name is inadvertently extruded. To simplify the notation, we let the composition operator \| bind stronger than the operator \triangleright , and the latter stronger than replication !. Every call to `cell()` gives rise to an object of type `cell`. The names `get` and `set` are blocked only positively, i.e. the object accepts messages headed by `get` and `set`, but will not transmit such messages. Hence, once they have entered the context of an agent representing a cell object, `get` and `set` messages are not allowed to leave the context, being thus forced uniquely to react within it. By contrast, a state name is wholly blocked, since its meaning is internal to the object. A state name is declared as a restricted name, and may thus be regarded as encapsulated within the object where it occurs. The object has the responsibility of updating its state, which is done by the emission of an agent of type `contents(\nu)`.

The channel `k` in a `get` message is used for returning the value of the method invocation. Normally the identifier of the requesting object should be used for the reply. In this case `get` would take two arguments, `k` and `i`, where `i` is the identifier name of the requesting object. The reply should consist of the message `i(k,x)`, instead of `k(x)`. Nevertheless, we prefer to use the latter variant, which may always be transformed to the former.

We may also let `contents` be initialised explicitly during object creation:
Usually objects are given names or identifiers, and methods should be invoked only via these identifiers. We can do this by blocking `get` and `set` totally, and by adding a new argument `k` to `cell` for returning the unique name of the object, which appears blocked only positively by the agent representing the object:

```
!cell(i, k) ▷ (ν contents, n)!
( k(n) | contents(i) |
  ! n(m, arg) ▷ m(arg)
  ! (get(k) | contents(x) ▷ k(x) | contents(x)) |
  ! (set(y) | contents(x) ▷ contents(y)) |
) \ {get, set, contents, n^+}
```

We call the agent

```
! n(m, arg) ▷ m(arg)
```

a request handler.

What happens if the first argument to `n` above is neither `set` nor `get`? The issue of unbound names is complex. In the case of `cell`, the result is a message that will be inadvertently transmitted to the environment. Any message headed by a name that is not known to the object is regarded as a message that should be transmitted to the environment. We cannot in general rule out this possibility, since in many cases this might be the intention. Here we could avoid this behaviour by testing the argument for equality with either `set` or `get`:

```
!cell(i, k) ▷ (ν contents, n)!
( k(n) | contents(i) |
  ! (n(m, arg) ▷ m(arg) | [m = get] get(arg) | (m = set) set(arg) | 0) |
  ! (get(k) | contents(x) ▷ k(x) | contents(x)) |
  ! (set(y) | contents(x) ▷ contents(y)) |
) \ {get, set, contents, n^+}
```

This is the standard way to define method calls inside objects. A public name denoting the method is sent to the object, which subsequently invokes the corresponding method, usually via an internal name. This procedure works well as long as only first-order values are exchanged. In the presence of agent mobility, where agents are allowed to move inside an object and for accessing its resources, this scheme does not work.

Suppose now that the method `set` in `cell` synchronises along a received channel as an acknowledgement to the message sender that the object has been updated. In this case, the number of arguments in the methods of the class may vary. A possible solution is to let the request handler be composed of several agents:

```
!cell(i, k) ▷ (ν contents, n)!
( k(n) | contents(i) |
  ! (n(m, a) ▷ m(a))
  ! (n(m, a1, a2) ▷ m(a1, a2))
  ! (get(k) | contents(x) ▷ k(x) | contents(x)) |
  ! (set(y, k) | contents(x) ▷ contents(y) | k())
) \ {get, set, contents, n^+}
```
A better solution is to allow for a variable number of arguments. This may be regarded as syntactic sugar, or may be incorporated in the syntax of the language. We choose here the former option, and use the notation \( m, \text{args} \) to denote a list of arguments whose first element is \( m \), followed by the arguments in \( \text{args} \), a (possibly empty) name vector:

\[
\langle \text{cell}(i, k) \rangle \Downarrow (\nu \text{contents}, n)(
\begin{align*}
&k[n] \mid \text{contents}(\bar{i}) \\
&! (n(m, \text{args}) \Rightarrow \text{m}(\text{args})) \\
&! \langle \text{get}(k) \mid \text{contents}(x) \Rightarrow k[x] \mid \text{contents}(\bar{x}) \rangle \\
&! \langle \text{set}(y, k) \mid \text{contents}(x) \Rightarrow \text{contents}(y) \mid k' \rangle \\
&\setminus \{ \text{get, set, contents, } n^+ \}
\end{align*}
\]

We now redefine \texttt{cell} as follows:

\begin{verbatim}
Class \texttt{cell(i)}
  var \texttt{contents} := i;
  method \texttt{get}() begin return \texttt{self.contents} end;
  method \texttt{set}(n) begin \texttt{self.contents} := n end;
  method \texttt{reset}(n) begin \texttt{self.set}(0) end;
end
\end{verbatim}

The new definition of \texttt{cell} includes in its interface the method \texttt{reset}, absent in the former version. An invocation of \texttt{reset} sets the value of \texttt{contents} to 0. We could have defined \texttt{reset} as

\begin{verbatim}
method \texttt{reset}(n) begin \texttt{self.contents} := 0 end,
\end{verbatim}

but note that the methods of a class may be overridden in subclasses, and thus the latter version could behave differently if \texttt{set} is itself redefined in \texttt{cell} or a subclass of \texttt{cell}.

In our calculus, we may encode \texttt{cell} as follows:

\[
\langle \text{cell}(i, k) \rangle \Downarrow (\nu \text{contents}, n)(
\begin{align*}
&k[n] \mid \text{contents}(\bar{i}) \\
&! (n(m, \text{args}) \Rightarrow \text{m}(\text{args})) \\
&! \langle \text{get}(k) \mid \text{contents}(x) \Rightarrow k[x] \mid \text{contents}(\bar{x}) \rangle \\
&! \langle \text{set}(y, k) \mid \text{contents}(x) \Rightarrow \text{contents}(y) \mid k' \rangle \\
&\setminus \{ \text{get, set, contents, } n^+ \}
\end{align*}
\]

We can treat a local method call as a message from the object to itself:

\[
\langle \text{cell}(i, k) \rangle \Downarrow (\nu \text{contents}, n)(
\begin{align*}
&k[n] \mid \text{contents}(\bar{i}) \\
&! (n(m, \text{args}) \Rightarrow \text{m}(\text{args})) \\
&! \langle \text{get}(k) \mid \text{contents}(x) \Rightarrow k[x] \mid \text{contents}(\bar{x}) \rangle \\
&! \langle \text{set}(y, k) \mid \text{contents}(x) \Rightarrow \text{contents}(y) \mid k' \rangle \\
&\setminus \{ \text{get, set, contents, } n^+ \}
\end{align*}
\]

\textbf{Points}

We turn to another example. The class \texttt{Point} defines objects whose internal state consists of two fields, \texttt{xcoor} and \texttt{ycoor}, and whose interface includes two methods \texttt{getx} and \texttt{gety} to access the fields, and a method \texttt{equal} that takes as argument a
point \( p \) and returns the value \( \text{true} \) if the coordinates of \( p \) are the same as its own coordinates, otherwise \( \text{false} \):

**Class** \( \text{Point}(x:\text{Int}, y:\text{Int}) \)

\[
\text{var} \ xcoor := x, \ ycoor := y;
\]

\[
\text{method} \ \text{get}_x() := \text{return} \ xcoor;
\]

\[
\text{method} \ \text{get}_y() := \text{return} \ ycoor;
\]

\[
\text{method} \ \text{equal}(p: \ \text{point}) \begin{align*}
\text{return} \ (\text{get}_x(\text{self}) = \text{get}_x(p) & \land \text{get}_y(\text{self}) = \text{get}_y(p) \end{align*};
\end{align*}
\]

In CRCHAM this can be defined as follows:

\[
\begin{align*}
\begin{cases}
\text{point}(x, y, k) \triangleright \ (\nu \ xcoor, \ ycoor, \ n)(
\begin{align*}
( & k[n] \mid xcoor(x) \mid ycoor(y) \\
\text{!} \ n(\text{m.args}) \triangleright m[\text{args}] \\
\text{!} \ \text{get}_x(k) \triangleright (k[x] \mid xcoor(x)) \\
\text{!} \ \text{get}_y(k) \triangleright (k[y] \mid ycoor(y)) \\
\text{!} \ \text{equal}(p, k) \triangleright (\text{get}_x, \ k_x)(\text{get}_y, \ k_y) \triangleright \ (\text{get}_x, \ k_x)(\text{get}_y, \ k_y) \\
\{k_x(p_x) \mid k_y(p_y)\} \triangleright [p_x = x] \{[[p_y = y] \ k(t) \ k(f)] \\
\} \ \{\text{get}_x, \ \text{get}_y, \ xcoor, \ ycoor, \ n+\})
\end{align*}
\end{cases}
\end{align*}
\]

We assume the existence of names \( t \) and \( f \) denoting the values \( \text{true} \) resp. \( \text{false} \). Note the use of the join pattern

\[
k_x(p_x) \mid k_y(p_y)
\]

which fires only once.

**Exporting Messages**

There are occasions when an agent wants to export a message that is bound or blocked within the environment of the agent. In order to do this an object may be defined as wrapped inside another environment whose function is simply to relay messages intended to be exported. Thus, assume \( P \) is any agent that wants to send the message \( x(y) \) to an agent \( Q \), but \( P \) is blocked by \( x \), as in the system

\[
(P \backslash x) \parallel Q.
\]

We may “wrap” the agent \( P \backslash x \) in an environment relaying messages of type \( \text{out}(m.args) \) as follows:

\[
((P \backslash x) \parallel \text{out}(m.args) \triangleright m[\text{args}]) \backslash \{\text{out}\}
\]

Thus, instead of sending the message \( x(y) \), the agent \( P \) may send \( \text{out}(x, y) \).

Below, in order not to clutter up the notation unnecessarily, we will assume that agents are wrapped in a relaying environment whenever a message \( \text{out}(\text{widetilde{idea}}) \), i.e., a message whose subject is \( \text{out} \), is emitted.

**5.3.2 Encoding Common Data Structures**

We show in this section the encoding of some common data structures in CRCHAM. The expression \( \text{if} \ldots \text{then} \ldots \text{else} \) is used below as syntactic sugar for if-then-else constructs.
Dictionaries

A dictionary is a data structure associating names and values. A dictionary may be encoded as an object that must perform at least two operations: associating a key and a value, and finding the current value associated with a particular key. A dictionary may be encoded in CRCHAM as follows:

```
\(\text{dictionary}(\tau) \triangleright (\nu \text{find}, \text{nil, head, add, n})(\gamma(n)) \mid \text{head}(\text{nil}) \mid (!\text{(m.args)} \triangleright \text{m(args)})
```

```
\[\begin{align*}
&!\text{(put}(k,v) \mid \text{head}(e)\triangleright \begin{cases}
  &\begin{cases}
    &\text{if } [e = \text{nil}] \text{ then } (\nu e')(\text{head}(e') \mid e'(k,v,\text{nil}))
    
    &\begin{cases}
      &\text{else } (\text{head}(e) \mid \text{add}(e,k,v))
    \end{cases}
  \end{cases}
  
  &\begin{cases}
    &\text{if } [k = k'] \text{ then } r(v)
    
    &\begin{cases}
      &\text{else } \text{find}(e',k,r)
    \end{cases}
  \end{cases}
  
  &\begin{cases}
    &\text{else if } [e' = \text{nil}] \text{ then } (\nu \text{next})(e(k,v,\text{next}) \mid \text{next}(k,v,\text{nil}))
    
    &\begin{cases}
      &\text{else } e(k',v',\text{nil}) \mid \text{add}(e',k,v))
    \end{cases}
  \end{cases}
\end{cases}
\end{align*}\]
```

A dictionary is implemented here as a linked list of elements of shape \(e(k,v,nxt)\), where \(e\) is the name of the element, \(k\) and \(v\) are the key resp. value corresponding to this element, and \(nxt\) is the next element in the linked list, if there is one, otherwise it is \(\text{nil}\), an internal name denoting the null element. The head of the list is represented by \(\text{head}(e)\), where \(e\) is the name of the first element of the list. The interface of the dictionary consists of the methods \text{get} and \text{put}. A message of type \text{put}(k,v) results in the addition of a new pair \((k,v)\) to the end of the list, on condition that there is no element with the same key in the list, otherwise the value of the latter is simply updated to \(v\). This is done by searching the list for an element with the same key, starting from the head of the list. If this element is \(\text{nil}\), i.e. if the list is empty, a fresh name \(e'\) is created and put at the beginning of the list. At the same time, \(e'(k,v,\text{nil})\) is emitted. Otherwise the first element in the list is searched by invoking \text{add}(e,k,v). This method searches the key \(k'\) and value \(v'\) associated with \(e\), and update its value to \(v\) if \(k = k'\). Otherwise, if the next element is \(\text{nil}\), a new entry \(\text{next}(k,v,\text{nil})\) is created, where \(\text{next}\) is a fresh name, and put at the end of the list. If the next element is not \(\text{nil}\), add is invoked recursively with \text{add}(\text{e'},k,v).

A value in the list with key \(k\) is looked up via \text{get}(k,r), where \(r\) is the reply channel. This is done by accessing the first element \(e\) of the list, and then by invocation of the private method \text{find}(k,r,e). If the end of the list has been reached, i.e. if \(e =\text{nil}\), then an error will be returned. We assume that a special name \text{error} is reserved for this purpose. Otherwise the element represented by \(e(k,v,\text{next})\) is accessed. If \(k = k'\), then \(v\) is returned; otherwise the next element of the list, \(\text{next}\), is recursively processed, and so on until the end of the list has been reached.
Records

A record can be seen as a dictionary with a fixed number of fields. A record with fields \( r_1, \ldots, r_n \) may be defined as follows:

\[
!record(k) \triangleright (\nu n, nil)(
(k[n] \mid r_1(nil) \mid \ldots \mid r_n(nil))
\]

\[
!m(args) \triangleright (\text{if } m = \text{get} \text{ or } m = \text{put} \text{ then } m(args))
\]

Note that in this definition only correct messages are accepted, since otherwise the state of record could become corrupted. An alternative would be to define private names for the fields of the record, and to keep a list with the correspondence between public and private names. The expression \( \text{if } [x_1 = y_1] \text{ or } [x_2 = y_2] \text{ then } P \) is syntactic sugar for \( \text{if } [x_1 = y_1] \text{ then } (\text{if } [x_2 = y_2] \text{ then } P \text{ else } 0) \text{ else } 0 \).

Arrays

We assume the existence of objects representing the natural numbers, which may accept messages representing the arithmetic operations, cf. Smalltalk-80 [56]. Natural numbers can be encoded in CRCHAM, but we omit it. If \( n \) and \( m \) are natural numbers, \( n + m \) can be represented by a message of type \( n(\text{plus}, n, r) \), where \( r \) is the reply channel that will return a name representing the natural number \( n + m \). Other possible messages are \( \text{equal} \), \( \text{less} \), \( \text{greater} \), \( \text{leq} \) (less or equal), \( \text{geq} \) (greater or equal), and the operations \( \text{suc} \), \( \text{plus} \), \( \text{minus} \), \( \text{div} \), \( \text{times} \), etc. We assume the first index of an array is always \( 1 \), and that the message \( \text{array}(n, r) \) creates an array indexed by \( 1, 2, \ldots, n \), where the initial value for each index is \( \text{nil} \).

\[
!\text{array}(n, r) \triangleright (\nu a, \text{head}, \text{nil}, \text{init})(
(r'[a] \mid \text{init}(n) \mid \text{head}(\text{nil}) \mid (a'm(args) \triangleright m(args))
\]

\[
\langle(v e)(\text{head}(e) \mid \text{init}(e, 1, n))
\]

\[
!\text{init}(e, m, n) \triangleright
\begin{cases}
(\nu r)(m(\text{less}, n, r) \triangleright r(k) \triangleright \\
\quad \text{if } [b = \text{true}] \text{ then } (v e')(m(\text{succ}, r) \triangleright r(k') \triangleright \text{init}(e', k, n))
\quad \text{else } c(m, n, \text{nil})
\end{cases}
\]

\[
!\text{put}(i, v) \mid \text{head}(e) \triangleright \text{head}(e) \mid \text{change}(e, i, v)
\]

\[
!\text{get}(i, r) \mid \text{head}(e) \triangleright \text{head}(e) \mid \text{find}(e, i, r)
\]

\[
!\text{find}(e, i, r) \triangleright \begin{cases}
[v = \text{nil}] \text{ then } r(\text{error})
\quad \text{else } e(i', v, \text{next}) \triangleright (e(i, v, e') \mid
\quad \text{if } [i = i'] \text{ then } r(v) \text{ else } \text{find}(e, i, r))
\end{cases}
\]

\[
!\text{change}(e, i, v) \triangleright e(i', v, e') \triangleright
\begin{cases}
[i = i'] \text{ then } e(i', v, e')
\quad \text{else } e(i', v, e') \mid [e' = \text{nil}] \text{ then } 0
\quad \text{else } \text{change}(e, i, v)
\end{cases}
\]

\[
\langle\text{init}, \text{nil}, \text{head}, \text{change}, \text{put}, \text{get}, \text{find}, a^+ \rangle
\]
Lists, Sets and Bags

We can implement a set as a list without repetitions. We endow it with the capability to both add and remove elements. The interface consists of \texttt{put}(v) for adding a new element \( v \) to the set; \texttt{remove}(v) for deleting an element \( v \) from the set, if there is one; and \texttt{member}(v,r), where \( r \) is a reply channel, for checking whether \( v \) belongs to the set. We assume the existence of predefined names \texttt{true} and \texttt{false} denoting the boolean values. Elements of the set are represented by \( e(v,e') \), where \( e \) represents the element itself, \( v \) its value, and \( e' \) the next element in the list. The head of the list is represented by an agent of type \texttt{head}(e). The definition of \texttt{set} is the following:

\[
\begin{align*}
\texttt{set}(r) & \triangleright (\nu \texttt{nil}, \texttt{head}, \texttt{add}, \texttt{eliminate}, \texttt{belongs}, n)(
\begin{array}{l}
\texttt{r}(n) \mid \texttt{head}(\texttt{nil}) \mid (n<\texttt{m.args}) \triangleright m<\texttt{args})
\end{array})
\end{align*}
\]

\[
\begin{align*}
\texttt{add}(e,v) & \triangleright e(v,e') \triangleright \texttt{if } [v = v'] \texttt{ then } (\nu e'')((\texttt{head}(e') \mid e''v,\texttt{nil})
\end{align*}
\]

\[
\begin{align*}
\texttt{remove}(v) & \triangleright \texttt{head}(e) \triangleright \texttt{if } [v = v'] \texttt{ then } \texttt{head}(e')
\end{align*}
\]

\[
\begin{align*}
\texttt{eliminate}(e,v,e') & \triangleright e(v',e'') \triangleright \texttt{if } [v = v'] \texttt{ then } e(v',e'')
\end{align*}
\]

\[
\begin{align*}
\texttt{belongs}(v,e,r) & \triangleright (\nu e'')((\texttt{head}(e'') \mid e''v,\texttt{nil})
\end{align*}
\]

A list can be implemented similarly, allowing repetitions, and is slightly simpler to encode. Bags can be encoded as lists, as well as buffers.

Queues

A queue resembles a list in which elements are always put at the end of the list and removed from the head of the list. The interface of a queue consists of the method names \texttt{enqueue} and \texttt{dequeue}. For efficiency, the last element \( e \) of a queue is kept by the agent \texttt{tail}(e).

\[
\begin{align*}
\texttt{enqueue}(v) & \triangleright (\nu \texttt{head},\texttt{tail}, \texttt{nil}, n)(
\begin{array}{l}
\texttt{r}(n) \mid (n<\texttt{m.args}) \triangleright m<\texttt{args}) \mid \texttt{head}(\texttt{nil}) \mid \texttt{tail}(\texttt{nil})
\end{array})
\end{align*}
\]

\[
\begin{align*}
\texttt{enqueue}(v) & \triangleright \texttt{head}(e) \mid \texttt{tail}(e') \triangleright \texttt{if } [e = n] \texttt{ then } (\nu e'')((\texttt{head}(e'') \mid \texttt{tail}(e'') \mid e''v,\texttt{nil})
\end{align*}
\]

\[
\begin{align*}
\texttt{dequeue}(r) & \triangleright \texttt{head}(e) \mid \texttt{tail}(e') \triangleright \texttt{if } [e = n] \texttt{ then } \texttt{head}(e) \mid \texttt{tail}(e') \mid \texttt{r}(\texttt{error})
\end{align*}
\]
else \( e(e') \rightarrow (head(e') | \text{if } [e = e'] \text{ then } tail(e') \text{ else } tail(e')) \)
\[
\{\text{nil, head, tail, enqueue, dequeue, } n^+ \}
\]

A Priority Queue

In [77], a priority queue is defined in the language \( \pi\lambda \). It consists of a queue of objects of class \( Priq \). It delivers and removes its smallest value via a method \( \text{remove} \), and accepts new values via a method \( \text{add} \). The state of each object is determined by the value it represents, assumed to be a natural number, and a link to the next element. The definition of the class in \( \pi\lambda \) is as follows:

\( Priq \) class
\[
\begin{align*}
\text{vars} & \quad m : [N] \leftarrow \text{nil}; l : \text{private ref}(Priq) \leftarrow \text{nil} \\
\text{add}(e : N) & \quad \text{method} \\
& \quad \text{return} \\
& \quad \quad \text{if } m = \text{nil} \text{ then } (m \leftarrow e; l \leftarrow \text{new } Priq) \\
& \quad \quad \text{elif } m < e \text{ then } l.add(e) \\
& \quad \quad \text{else } (l.add(m); m \leftarrow e) \\
\end{align*}
\]

\( \text{remove}() \) method \( r : N \\
\text{return } m \\
\quad \text{if } m \neq \text{nil} \text{ then } m \leftarrow l.\text{remove}() \\
\quad \quad \text{if } m = \text{nil} \text{ then } l \leftarrow \text{nil} \\
\quad \quad \text{fi} \\
\text{fi}
\]

This class may be encoded in CRCHAM as follows:

\[
\begin{align*}
\text{priq} & \quad \text{> (nu add, rm, m, n)} \\
(r(m)) & \quad \text{> (m(args)) | m(nil) | (\text{nil})} \\
\quad & \quad \text{> (add(r) | m(v') | (p) \triangleright r(v))} \\
\quad & \quad \quad \quad \text{if } [v' = \text{nil}] \text{ then } m(v) | (\text{nu r})(\text{priq}(v')) | r(v') \triangleright (p(v')) \\
\quad & \quad \quad \quad \text{else } (\text{nu r})(p(v, v', v, v') | r(v) \triangleright (p(v))) \\
\quad & \quad \quad \quad \quad \quad \text{if } [b = \text{true}] \text{ then } (p(\text{add}, v) | m(v')) \\
\quad & \quad \quad \quad \quad \quad \text{else } (p(\text{add}, v') | m(v))) \\
\quad & \quad \quad \quad \text{fi} \\
\quad & \quad \quad \text{fi} \\
\quad & \quad \text{fi} \\
\quad & \quad \text{fi} \\
\quad & \quad \text{fi} \\
\end{align*}
\]

\[
\begin{align*}
\{\text{nil, add, rm, m, n}^+ \}
\end{align*}
\]

5.3.3 Subclasses

Classes may be viewed as templates for objects, or code that can be reused in the definition of subclasses. We define here a language for defining class templates. We illustrate the procedure by analysing one example, the class \textit{cell} defined earlier:
\[ \text{cell}(i, k) \triangleright (\nu \text{ contents}, n) \]
\[ \quad \begin{align*}
& (k[n]) | \text{contents}(z) | \\
& ![m(\text{args}) \triangleright m(\text{args})] \\
& ![\text{get}(k) | \text{contents}(x) \triangleright k[x] | \text{contents}(z)] | \\
& ![\text{set}(y, k) | \text{contents}(x) \triangleright \text{contents}(y) | k[j]] | \\
& ![\text{reset}(k) \triangleright \text{set}(0, k)]
\end{align*} \]

The class definition of cell includes the state names (contents), the initialiser, the request handler, the class interface (get, set, reset), and the definition of methods for the class interface. Each class has a special policy concerning the visibility of its fields and methods. In our example we assume that state names are purely private, whereas names representing methods are public but are interpreted locally. This policy will be enforced by the blocking policy associated with each class.

We make some definitions in a rather informal notation, which may be viewed as a metalanguage for CRCHAM. The expression \( \langle x_1, \ldots, x_n \rangle \) denotes a vector with elements \( x_1, \ldots, x_n \). For simplicity, we overload the meaning of this construct and let it also denote a set when it occurs in a context where a set is expected. Also, for any set \( S = \{x_1, \ldots, x_n\} \), we write \( (\nu S) \) for \( \langle \nu x_1, \ldots, x_n \rangle \). The expression \( v_1 \triangleright v_2 \) denotes the vector \( v_2 \) appended to \( v_1 \). The definitions are the following:

\[
\begin{align*}
\text{icell} & ::= \langle \text{get, set, reset} \rangle \\
\text{Scell} & ::= \langle \text{contents} \rangle \\
\text{Bcell} & ::= \text{icell} \cup \text{Scell} \cup \langle n^+ \rangle \\
\text{cell\_handler} & ::= ![n(m(\text{args}) \triangleright m(\text{args})] \\
\text{cell\_state} & ::= \text{contents}(i) \\
\text{cell\_methods} & ::= ![\text{get}(k) | \text{contents}(x) \triangleright k[x] | \text{contents}(z)] | \\
& \quad ![\text{set}(y, k) | \text{contents}(x) \triangleright \text{contents}(y) | k[j]] | \\
& \quad ![\text{reset}(k) \triangleright \text{set}(0, k)]
\end{align*}
\]

We use names in bold style as a notational convenience to indicate that these names belong to the metalanguage and should normally be substituted by the corresponding class interface names. We assume each name in CRCHAM has a corresponding name in the metalanguage. Also, if \( S \) is a set of names in CRCHAM, we write \( S \) in bold style to denote the corresponding set of names in the metalanguage. A name in bold style should be substituted by the corresponding interface name of the class in the class definition, but may be instantiated to another (freshly created) name if the method it represents is overridden in a subclass definition. Note that only occurrences of the interface names in the join patterns are in bold style. The reason is that thereby we obtain dynamic or late binding in invocations of super methods by subclass objects, as will become apparent below. Observe the occurrence of the name set in plain style in the “body” of reset, indicating that this occurrence should not be redefined in a subclass.

icell denotes the interface of the class cell, and Scell the names representing the state names. Bcell is the set of the names blocked by the class, consisting of the interface names, the state names, and the object identifier, which is blocked only positively. The cell\_handler represents the request handler of the class abstracted over the object identifier. The cell\_state denotes the initial state of an object, abstracted over both the state name and its initial value. Finally, the cell\_methods corresponds to the definition or body of the methods of the class. Now we may define the class cell as follows:
\(\text{cell}(i, k) \triangleright (\nu \text{Cell}, n)\) \\
\(\text{(k(n) | cell\_handler(n) | cell\_state | cell\_methods) \mid \text{Bcell}}\)

where

\[
\Phi ::= [\text{get} \to \text{get}, \text{set} \to \text{set}, \text{reset} \to \text{reset}]
\]

i.e. \(\Phi E\) means a textual substitution in expression \(E\) of names in the metalanguage to names in CRCHAM. For simplicity of notation we also write

\[
\Phi ::= \text{Icell} \rightarrow \text{Icell}
\]

We may define a \text{cell\_body} comprising the state and methods of the class \text{cell}:

\[
\text{cell\_body} ::= \text{cell\_state} | \Phi \text{cell\_methods}
\]

The definition of \text{cell} now becomes:

\[
\text{cell}(i, k) \triangleright (\nu \text{Cell}, n)\text{(k(n) | cell\_handler | cell\_body)}
\]

We declare now \text{RCell}, a subclass to \text{cell}, as follows:

\begin{verbatim}
subclass \text{RCell}(i) of \text{cell}2
  var backup := 0;
  override set(x) begin self.backup := self.contents;
    self.contents := x end;
  method restore() begin
    self.contents := self.backup end;
end
\end{verbatim}

In CRCHAM we may define \text{RCell} as follows:

\[
\text{rCell}(i, k) \triangleright (\nu \text{backup, contents, set'}, n)\text{(}
\(\text{k(n) | backup(0) | rCell\_handler}\)

\[
\begin{array}{l}
(\text{set}(x,k) \mid \text{contents}(y) \mid \text{backup}(z) \triangleright \text{k}() | \text{contents}(z) | \text{backup}(y)) | \\
(\text{rStore}() \mid \text{contents}(y) \mid \text{backup}(z) \triangleright \text{k}() | \text{contents}(z) | \text{backup}(z)) | \\
\Phi' \text{cell\_body} | \\
\text{(Bcell} \cup \{\text{set', rstore, backup}\})
\end{array}
\]

where

\[
\Phi' = \Phi[\text{set} \to \text{set'}]
\]

i.e. \(\Phi'\) is the same substitution as \(\Phi\) except that \text{set} is substituted by \text{set'} instead. In general, if \(\Phi_1\) and \(\Phi_2\) are two substitutions, we write \(\Phi_1 \Phi_2\) to mean \(\Phi_2\) appended to \(\Phi_1\). If a name occur twice at the left-hand side of a substitution, the rightmost occurrence has precedence and overshadows the other occurrences.

An eventual invocation of the superclass version of the overridden name \text{set} may take place as follows. Suppose we redefine \text{RCell} slightly:
subclass \texttt{reCell} of \texttt{cell2}

\begin{verbatim}
var backup := 0;
override set(x) begin self.backup := self.contents;
super.set(x) end;
method restore() begin
    self.contents := self.backup end;
end
\end{verbatim}

Now the method \texttt{set} in the subclass invokes its superclass version. This may be translated as

\begin{verbatim}
\texttt{!reCell(i,k) }\triangleright (\forall \texttt{backup, contents, set, n})(
    (k,n) | \texttt{backup(n)} | 
    \texttt{reCell_handler(n)} | 
    \texttt{!(set}(x,k) | \texttt{contents(y)} | \texttt{backup(z)}) \triangleright \texttt{set}(x,k) | \texttt{contents(y)} | \texttt{backup(z)} | 
    \texttt{!(restore()} | \texttt{contents(y)} | \texttt{backup(z)}) \triangleright k() | \texttt{contents(z)} | \texttt{backup(z)} | 
    \texttt{\exists'\ cell\_body} 
    ) \setminus\{\texttt{cell, set, restore, backup, n+}\}
\end{verbatim}

The reason why only the occurrence of \texttt{set} in the reaction pattern is substituted, is to allow for dynamic dispatch of methods invoked within the body of a the subclass method.

We make the following definitions:

\begin{align*}
\texttt{Srecell}^+ & := \{\texttt{backup}\} \\
\texttt{Srecell} & := \texttt{Srecell}^+ \cup \texttt{Scell} \\
\texttt{Irecell}^+ & := \{\texttt{restore}\} \\
\texttt{Irecell} & := \texttt{Irecell}^+ \cup \texttt{Icell} \\
\texttt{Orecell} & := \{\texttt{set'}\} \\
\texttt{Breccel} & := \texttt{Irecell} \cup \texttt{Srecell} \cup \texttt{Orecell} \cup \{n^+\} \\
\texttt{reCell\_handler} & := \{!(m.\texttt{args}) \triangleright m(\texttt{args})\} \\
\texttt{reCell\_state}^+ & := \{\texttt{backup}(0)\} \\
\texttt{reCell\_methods}^+ & := \{!(\texttt{set}(x,k) | \texttt{contents}(y) | \texttt{backup}(z)) \triangleright k() | \texttt{contents(z)} | \texttt{backup(z)} | 
    \texttt{!(restore()} | \texttt{contents(y)} | \texttt{backup(z)}) \triangleright k() | \texttt{contents(y)} | \texttt{backup(y)} | 
    \texttt{\exists'\ cell\_body} 
    | \texttt{\exists'\ cell\_body}^+\}
\texttt{reCell\_body}^+ & := \{\texttt{reCell\_state}^+ | \texttt{reCell\_methods}^+\} \\
\texttt{reCell\_body} & := \{\texttt{\exists'\ cell\_body} | \texttt{\exists''\ cell\_body}^+\}
\end{align*}

where

\begin{equation*}
\texttt{\exists''} := \texttt{IreCell} \rightarrow \texttt{IreCell}
\end{equation*}

The notation \texttt{Srecell}^+ ( \texttt{Irecell}^+ , etc.) is intended to indicate that this is not the complete state names of \texttt{reCell}, but only the part that has been added to \texttt{Scell}. The set of state names of the class \texttt{reCell} consists of \texttt{Scell} \cup \texttt{Srecell}^+. We may now define \texttt{recell} as follows:

\begin{verbatim}
\texttt{!reCell(i,k) }\triangleright (\forall \texttt{Srecell, Orecell, n}(k,n) | \texttt{reCell\_handler} | \texttt{reCell\_body}\setminus(\texttt{BreCell, n+})
\end{verbatim}

We now formalise this procedure. Let \( c \) be any class, and \( sc \) a subclass of \( c \). We need the following definitions:

\begin{verbatim}
\end{verbatim}
\( I_c := \text{c's interface} \)
\( \Phi_c := I_c \rightarrow I_c \)
\( S_c := \text{c's states names} \)
\( \bar{i}_c := \text{c's initialisation names} \)
\( \text{state}_c := \text{agents defining initial state} \)
\( B_c := I_c \cup S_c \)
\( \text{methods}_c := \text{methods for c with metalevel names} \)
\( \text{handler}_c := !(n(m(args) \Rightarrow m(args))) \)
\( \text{body}_c := \text{state}_c \cup \Phi \text{methods}_c \)

With these definitions, the class \( c \) may be defined as
\[
!c(\bar{i},k) \Rightarrow (\nu S_c, n)((k(n) \mid \text{handler}_c, \text{body}_c) \backslash \{B_c, n^+\})
\]

For the subclass \( sc \) of \( c \) we need the following definitions:

\( I^+_sc := \text{interface names added to } I_c \text{ in } sc \)
\( I_{sc} := I_c \cup I^+_sc \text{ (sc's total interface)} \)
\( S^+_sc := \text{new state names for } sc \)
\( S_{sc} := S_c \cup S^+_sc \)
\( O_{sc} := \text{fresh names for overridden names} \)
\( B_{sc} := I^+_sc \cup S_{sc} \cup O_{sc} \)
\( \bar{i}_{sc} := \bar{i}_c \)
\( \Phi_{sc} := \Phi_c \Rightarrow I_{sc} \)
\( \Phi^+_{sc} := \Phi_{sc} \mid \Phi_{sc} \Rightarrow O_{sc} \)
\( \text{handler}_{sc} := !(n(m(args) \Rightarrow m(args))) \)
\( \text{state}^+_{sc} := \text{added agents defining initial state} \)
\( \text{methods}^+_{sc} := \text{new methods for } sc \text{ with metalevel names} \)
\( \text{body}^+_{sc} := \text{state}^+_{sc} \cup \text{methods}^+_{sc} \)
\( \text{body}_{sc} := \Phi^+_{sc} \text{body}_c(S_c) \mid \Phi_{sc} \text{body}^+_{sc} \)

Then, the subclass \( cs \) may be defined as
\[
!c(\bar{i}_{sc},k) \Rightarrow (\nu S_{sc}, O_{sc}, n)((k(n) \mid \text{handler}_{sc}(n) \mid \text{body}_{sc}) \backslash \{B_{sc}, n^+\})
\]

### 5.4 Higher-Order CRCHAM

We extend the CRCHAM by allowing any terms in the language to occur as objects in messages. This amounts to allowing agents to be transmitted in communications. The syntax of higher-order CHCHAM, briefly HoCRCHAM, is an extension of the syntax of CRCHAM. In addition to a set \( N \) of names, it assumes the existence of a set \( X \) of process variables ranged by \( X, Y, \ldots \). An occurrence of a process variable in a message pattern is binding. An agent expression is closed if all occurrences of process variables are within the scope of a binding occurrence. A process variable is
bound by the nearest binding occurrence of the variable. A closed agent expression is called a process.

We let $V$ range over the union of the set of names and the set of agents, and $U$ over the union of the set of names and the set of process variables.

$$P \overset{\text{def}}{=} \text{agent expressions}$$

- $0$ null process
- $x(V)$ messages
- $X$ process variable
- $J \triangleright P$ reaction
- $P \parallel P$ parallel composition
- $P \setminus L$ blocking
- $(\nu x)P$ restriction
- $[x = y]PQ$ if-then-else
- $!P$ replication

$$J \overset{\text{def}}{=} \text{join-patterns}$$

- $x(U)$ message pattern
- $J \parallel J$ join of patterns

The definition of reaction (rtn), received (rcn) and free names (fn) are as follows

$$\text{rcn}(x(U)) \overset{\text{def}}{=} \{u : u \in U\}$$

$$\text{rcn}(J \parallel J') \overset{\text{def}}{=} \text{rcn}(J) \cup \text{rcn}(J')$$

$$\text{rtn}(x(V)) \overset{\text{def}}{=} \{x\}$$

$$\text{rtn}(J \parallel J') \overset{\text{def}}{=} \text{rtn}(J) \cup \text{rtn}(J')$$

$$\text{fn}(J \triangleright P) \overset{\text{def}}{=} \text{rtn}(J) \cup (\text{fn}(P) - \text{rcn}(J))$$

$$\text{fn}(x(V)) \overset{\text{def}}{=} \{x\} \cup \{u : u \in \text{fn}(V)\} \text{ for some } V \in \bar{V}$$

$$\text{fn}(P \parallel P') \overset{\text{def}}{=} \text{fn}(P) \cup \text{fn}(P')$$

$$\text{fn}([x = y]PQ) \overset{\text{def}}{=} \text{fn}(P) \cup \text{fn}(Q) \cup \{x, y\}$$

$$\text{fn}(P \setminus L) \overset{\text{def}}{=} \bar{L} \cup \text{fn}(P)$$

$$\text{fn}((\nu x)P) \overset{\text{def}}{=} \text{fn}(P) - \{x\}$$

$$\text{fn}(!P) \overset{\text{def}}{=} \text{fn}(P)$$

Here, $\bar{L} = \{x : x^+ \in L \lor x^- \in L\}$. A name $x$ is fresh in $P$ if $x \notin \text{fn}(P)$. An active context $C$ is defined as before.

The semantic rules are those of first-order CRCHAM in Section 5.2, except rules str-bout and str-bin which must be redefined as follows:

$$(\text{str-bout}) \quad (x(\bar{V}), S) \setminus L \rightarrow x(\bar{V}), S \setminus L \quad \text{if } x^+ \notin L$$

$$(\text{str-bin}) \quad x(\bar{V}), S \setminus L \rightarrow (x(\bar{V}), S) \setminus L \quad \text{if } x^- \notin L$$

We assume also that the substitution $J\sigma_{rcn}$ respects the type of name occurring in $J$, i.e. process variables are substituted by processes, and name variables by names.
5.5 Encoding Reflection in Smalltalk

5.5.1 Encoding Objects with Reflection in the Higher-Order CRCHAM

A reflective program is one that reasons about itself. Computational reflection [99, 98, 46] is the activity of doing computation about its own computation. A reflective system is a computational system which is able to perform reflective computation, i.e. to bring modifications to its behaviour as a result of its own computation. Such systems must incorporate structures representing aspects of itself and its own behaviour.

An example of limited reflective capability is given by the notion of meta-class introduced by Smalltalk-80 [56]. The idea behind meta-classes in Smalltalk-80 is that it should be possible to specify and manipulate the internal structure of a class. This is done in Smalltalk-80 by enabling meta-classes to override method lookup, which results in the redefinition of a method. In this section we will concentrate on this aspect of reflective computation in Smalltalk-80.

Object-oriented systems that support reflection by letting classes be first class objects contains features whose modelling in process calculi seems to require higher order constructs. Michael Papatheomas [133] suggested that

\[ \ldots \text{an object class may be modelled as an agent that stores the description of the agents corresponding to the class methods. Objects of the class receive the description of the method agents and execute them within a scope that restricts access to the object's instance variables. In this scope references to instance variables in the method agents are dynamically bound to the instance variables of the instance that executes them. Dynamic changes to classes and to the class inheritance hierarchy can be modelled by changing the description of method agents held by the agent representing the class or by changing in instance the value used to identify the agent acting as its class.} \]

This is basically what we intend to do next, using higher order CRCHAM as the target language to model a fragment of Smalltalk-80 and some examples in this language. The encoding we are going to show in the following section is intended mainly as a testing ground for assessing the expressive power of a higher order process calculus with dynamic binding of names.

5.5.2 Smalltalk-80

Meta-level objects, shortly metaobjects, are objects that hold reflective information, i.e. information about the definition, implementation and interpretation of other objects. A reflective object-oriented language supports the assessment of objects as well as alterations in their representation dynamically, i.e. at runtime. Metaobjects may be seen as the system’s self-representation. A language that supports the dynamic redefinition of objects is Smalltalk-80 [56].

In the Smalltalk-80 system, every object is an instance of a class. The instances of a class have a common message interface. A class describes both how operations available through the interface are carried out and the private memory belonging to its instances. Operations are described by methods. A method describes how an object performs one of its operations.

Classes are components in Smalltalk-80, and are thus represented by objects. The name of a class is globally shared; it is the identifier of the object representing the
There is no multiple inheritance in Smalltalk-80, and subclassing is strictly hierarchical. Class membership may thus not overlap, and every object is an instance of exactly one class.

The search for a method follows the superclass chain. When a message with a method invocation is received by the instance of a class, the methods in the receiver's class are searched first. If there is match, the methods in the superclass of the receiver's class are searched next. The search continues up the class hierarchy until a matching method is found. When a method is preceded by the pseudo-variable self, the search for the method begins in the instance's class. If a method is preceded by super instead, the search for the method begins in the superclass of the instance's class.

All Smalltalk system components are represented by objects, and all objects are instances of a class. Thus, the classes themselves must be represented by instances of a class. A class whose instances are classes is called a metaclass. Each class is an instance of its own metaclass. When a class is created, a new metaclass for it is automatically created. Like any other class, a metaclass contains the methods used by its instances. All metaclasses are instances of a class called Metaclass, and do not have class names. The messages of a metaclass support creation and initialisation of instances, as well as initialisation of class variables. Like other classes, a metaclass inherits from a superclass, and may add or redefine new methods.

5.5.3 A Smalltalk-80 Example: FinancialHistory

We present here a complete implementation description in Smalltalk-80 for a class FinancialHistory, according to [56]. For details about the syntax we refer to that work. We show here very briefly those features needed to understand the examples below.

Messages in Smalltalk may be unary messages or selector messages. The former are messages without arguments, consisting of an object identifier and a method name. The latter are messages with arguments, consisting of the object identifier followed by the name of the method invoked and its corresponding arguments. Each argument is preceded by a keyword followed by a semicolon. The expression \( \uparrow E \) denotes return \( E \), and the assignment of the value of the expression \( E \) to the variable \( x \) is denoted \( x \leftarrow E \). A block is an object that consists of a sequence of commands separated by a period and delimited by square brackets. These actions are activated only when the block receives the unary message value. Boolean expressions accept messages of type ifFalse:ifTrue:. An expression evaluating to true sends the message value to the first argument, otherwise to the second argument. A Dictionary is a data structure associating a key with an object. A dictionary accepts messages of type at: key put: value for including a key key with value value; of type includesKey: key which returns true if key occurs as one of the keys of the dictionary, otherwise false; and of type at: key which returns the value associated with the key key.

In a protocol description of a class, messages are grouped into categories according to some common functionality. This categorisation does not affect the operation of the class. A protocol lists messages with a comment about its functionality. The protocol description for FinancialHistory is the following:

transaction recording
receive: amount from: source  Remember that an amount of money, amount, has been received from source
spend: amount for: reason  Remember that an amount of money, amount, has been spent for reason

inquiries
  cashOnHand  Answer the total amount of money currently on hand.
incomes  Answer the total amount received from source, so far.
  expenditures  Answer the total amount spent for reason, so far.

initialisation
  initialBalance: amount  Begin a financial history with amount as the amount of money on hand.

We show next an implementation description of FinancialHistory, consisting of the class name, a declaration of the variables available to the instances, and the methods used by instances to respond to messages:

class name  FinancialHistory
superclass  Object
instance variable names  cashOnHand
                        incomes
                        expenditures

class methods

instance creation
  initialBalance: amount
  ↑ super new setInitialBalance: amount
  new
  ↑ super new setInitialBalance: 0

instance methods

transaction recording
  receive: amount from: source
    incomes at: source
    put: (self totalReceivedFrom: source) + amount .
    cashOnHand ← cashOnHand + amount

  spend: amount for: reason
    expenditures at: reason
    put: (self totalSpentFor: reason) + amount.
    cashOnHand ← cashOnHand - amount

inquiries
  cashOnHand
  ↑ cashOnHand

  totalReceivedFrom: source
  (incomes includesKey: source)
  ifTrue: [↑ expenditures at: reason]
  ifFalse: [↑ 0]
initialisation

setInitialBalance: amount
  cashOnHand ← amount.
  incomes ← Dictionary new.
  expenditures ← Dictionary new.

5.5.4 Encoding of FinancialHistory in CRCHAM

We now show a first approach to an encoding of FinancialHistory in higher order CRCHAM. It is assumed that class Dictionary and the integers, as explained in Section 5.3.2, are defined globally.

Basically, the organisation of the encoding follows the structure shown in Figure 5.1, which reproduces a similar figure in [50]. In this figure, a box denotes a class, and a circle denotes an instance of the class. Objects which are not classes are represented as circles located at the box representing the class. To each class (box) there corresponds a circle located within the box representing its metaclass, which in Smalltalk-80 are identified. However, in our encoding they are kept separated, each one consisting of a distinct agent with a blocking policy. The agent representing a box is called a class environment, while the agent representing a circle is the class itself seen as an object, and thus we call it the class object. We made this design decision because we believe that boxes and circles in Figure 5.1 play different roles, as suggested by this very picture. We thus decompose a class into an agent representing the bindings associated with the class, i.e. the method names or interface of the class, and an agent representing the class as an active object that is responsible for dealing with class messages. A box plays the role of a binding environment for the instances of the class, whereas the circles represent the class as an active object bearing the reflective capabilities of the class.

In our encoding, a class is represented as a context with a determined blocking policy, and consists roughly of two groups of agents, one for dealing with requests from the instances of the class, and a second for dealing with request from the class object. The former are basically requests for the body of a method, while the latter concerns the reflective capabilities of the classes. We study only requests for alterations in the existing methods, not for adding or removing methods. Furthermore, class methods and class variables may also be defined as agents located within the class environment. We let the state of the agents be represented by unrestricted names, since this simplifies the task of binding these names to the body of the methods and to the code of dynamically created subclasses. We could also have parametrised the method bodies over these names, which in this way could have been defined as fresh for each instance of the class.

Instances of a class occur within the class environment, and thus may send messages to the class by simply exporting them. Since the subject of those messages are the name of a method, these messages may be seen as moving up the class hierarchy in search for the first or innermost environment that binds the requested method. This procedure mirrors the search for a matching following the superclass chain in Smalltalk-80, as explained in Section 5.5.2. By contrast, the communication between a class environment and its metaclass may be viewed as transmissions along a privately shared channel.
Figure 5.1: Class organisation in Smalltalk-80
The FinancialHistory Class

The following definitions will be useful. They describe the body of a method parametrised on its identifier, \texttt{self}. Parametrisation of names are easily expressed in CRCHAM by prefixing an agent with a join pattern where the formal parameters appears as received names. We define a method body for each of the methods defined for the class \texttt{FinancialHistory}.

\begin{verbatim}
receiveBody ::= identifier(self) \triangleright
  (receive(amount, source) | cashOnHand(c))\triangleright
  (\nu r)(\{self|totalReceivedFor,r\} \triangleright
    r(total)\triangleright
    (\nu r')(total plus, amount, r')\triangleright
    r'(total)\triangleright
    d(put, source, total)) |
  (\nu r)(c\{plus, amount, r\} \triangleright
    r(c)\triangleright
    cashOnHand(c'))

spendBody ::= identifier(self) \triangleright
  (spend(amount, reason) | cashOnHand(c))\triangleright
  (\nu r)(\{self|totalSpentFor,r\} \triangleright
    r(total)\triangleright
    (\nu r')(total plus, amount, r')\triangleright
    r'(total)\triangleright
    e(put, reason, total)) |
  (\nu r)(c\{plus, amount, r\} \triangleright
    r(c)\triangleright
    cashOnHand(c'))

gETCHoNHandBody ::= identifier(self) \triangleright
  getCashOnHand(r) | cashOnHand(c) \triangleright
  r(c) | cashOnHand(c'))

totalReceivedFromBody ::= identifier(self) \triangleright
  (totalReceivedFrom(source, r) | incomes(d))\triangleright
  incomes(d) |
  (\nu r')(d(source, r')\triangleright
    r'(amount)\triangleright
    if [amount = error] then r(0) else r(amount))

totalSpentFromBody ::= identifier(self) \triangleright
  (totalSpentFor(reason, r) | expenditures(e))\triangleright
  expenditures(e) |
  (\nu r')(c(reason, r')\triangleright
    r'(amount)\triangleright
    if [amount = error] then r(0) else r(amount))

setInitialBalanceBody ::= identifier(self) \triangleright
  setInitialBalance(amount, r) \triangleright
  r() | cashOnHand(amount) |
  (\nu r)(dictionary(r) \triangleright
    r(d) \triangleright
    incomes(d) |
    (\nu r)(dictionary(r) \triangleright
      r(d) \triangleright
      expenditures(d)))
\end{verbatim}

We define now the encoding of an instance agent, parametrised over its name identifier.

121
FinancialHistoryInstance ::= setIdentifier(n) >
| (id|entifier(n) | ! (n(m, args) > (v r)(out(m, r) | (r(X) > (X | m(args))))) |
| !(super(m, args) > (v r)(out(sup, m, r) | (r(X) > (X | m(args))))) |
\/{(fhInterface ∪ fhState ∪ {super, n"})}

The agent FinancialHistoryInstance consists of the identifier name, a request handler for messages addressed to itself, and a handler for dealing with calls along super to the version of a method defined in the instance’s superclass.

The request handler is defined as

![n(m, args) > (v r)(out(m, r) | (r(X) > (X | m(args)))]

where n is the identifier of the object. After receiving a message m.args telling the instance to invoke method m with arguments args, the request handler exports the message m(r), as describe in Section 5.3.1. This is necessary since the instance agent itself binds the name m if this name denotes one of the methods of its interface. The agent representing an instance of a class is assumed to be located within its class environment, which also binds the name of methods defined for the class. The instance agent must thus look for the definition of this method within this environment, which is done by exporting the message m(r), where r is a reply channel. When the method agent has been received along r, it is simply activated together with the message m(args) with which it is supposed to react.

The super handler is similar, but instead of exporting the message m(args) it exports the agent super(m, args). When this message reaches the class environment, the latter simply exports m(r) it to the next environment up in the class hierarchy, which represents the superclass. The instance agent will finally receive the correct version of the method invoked.

The agent !identifier(n) may be seen as representing a constant that denotes the instance identifier. Since method bodies are prefixed by the join pattern identifier(self), the pseudo-variable self will be appropriately instantiated in the method body to the identifier name of the instance.

The interface and the state of the instance are defined as follows:

fhInterface ::= {receive, spend, getCashOnHand, totalReceivedFrom, totalSpentFor, setInitialBalance}
fhState ::= {cashOnHand, incomes, expenditure}

We define next the environment of FinancialHistory:

financialHistoryEnv ::=:
| (receive(receiveBody) | ![receive(X) | receive(r) > r(X) | receive(X)) |
| spend(spendBody) | ![spend(X) | spend(r) > r(X) | spend(X)) |
| getCashOnHand(getCashOnHandBody) | ![getCashOnHand(X) | getCashOnHand(r) > r(X) | getCashOnHand(X)] |
| totalReceivedFrom(totalReceivedFromBody) | ![totalReceivedFrom(X) | totalReceivedFrom(r) > r(X) | totalReceivedFrom(X)] |
| totalSpentFor(totalSpentForBody) | ![totalSpentFor(X) | totalSpentFor(r) > r(X) | totalSpentFor(X)] |
As we may observe, the class environment consists of the method bodies and of agents for dealing with requests from the instances of the class asking for the code of a method. Furthermore, there is one agent for relaying messages asking for the method of the superclass, which are exported to the next environment in the class chain.

This picture is not complete, since the reflective capabilities, e.g., the capability to change the body of a method, are still missing. In the next section we extend the FinancialHistoryEnv with agents that deal with messages received from the corresponding metaclass object.

### The FinancialHistory Metaclass

The FinancialHistory class is a metaclass with a unique instance. The metaclass organisation is symmetric to the class organisation, and the principles for encoding metaclasses are the same as those for encoding classes.

Creation of new instances of a class takes place via class messages, in our example to the unique instance of the metaclass of FinancialHistory. Class messages invoke class methods. One of the class methods for instance creation in FinancialHistory is initialBalance. This method invokes the method new of the Object class, the superclass of the FinancialHistory class. In Smalltalk-80 this gives rise to an invocation of a primitive method. For simplicity we assume that each class is endowed with an own version of new, whose body consists of a message sent to the respective class environment telling it to spawn a new object within its scope.

\[ \text{newBody} := \text{new}(r) \triangleright \text{class}(i) \triangleright i(\text{newObject}, r) \]

No other initialisation method exists in our simplified version.

A partial definition of the environment of the FinancialHistory class, without reflective capabilities, is the following:

\[
\text{financialHistoryClassEnv} :=
\begin{array}{l}
\text{new}(\text{newBody}) \\
\text{super}(m,r) \triangleright o\text{ut}(m,r) \\
\text{new}, \text{super}
\end{array}
\]

In the next section we extend this agent in order to deal with reflection.

The unique instance of FinancialHistory class is similar to the definition of instances of FinancialHistory, except that it contains an agent !class(i) as part of its state, where \(i\) denotes the channel used for communication with FinancialHistoryEnv:

\[
\text{financialHistoryClassInstance} :=
\begin{array}{l}
\text{!class}(i) \triangleright \text{identifier}(c) \\
\text{!c}(m,\text{args}) \triangleright (\nu r)(\text{out}(m,r) \triangleright r(X) \triangleright X \triangleright \text{m(args)})
\end{array}
\]
Adding Reflection to FinancialHistory

The FinancialHistory environment must be extended in order to deal with messages from its class. We assume here three type of messages: messages for creating a new object; messages for creating a new subclass; and messages for changing the body of a method. In the latter case the environment agent receives a message of kind change Method(M), where Method is the name of some of the methods defined for the class, and M is the new code for the method. The new definition of FinancialHistoryEnv is an extension of the earlier one with the following:

FinancialHistoryEnv ::= 

\[
\begin{align*}
\text{financialHistoryEnv} & := \\
\text{(\ldots)} \\
\text{\{newObject(r) \vdash (u,n)(FinancialHistoryInstance | setIdentifier(n) | (r \text{ setInitialBalance}, 0, r') \vdash r'() \vdash r(n) |)} \\
\text{\{newSubclass(SubClassEnv, t) \vdash (internal) | SubClassEnv \{\} |} \\
\text{\{receive(X) | changeReceive(Y) \vdash receive(Y) |} \\
\text{\{spend(X) | changeSpend(Y) \vdash spend(Y) |} \\
\text{\{getCashOnHand(X) | changeGetCashOnHand(Y) \vdash getCashOnHand(Y) |} \\
\text{\{totalReceiveFrom(X) | changeTotalReceiveFrom(Y) \vdash totalReceiveFrom(Y) |} \\
\text{\{totalSpentFor(X) | changeTotalSpentFor(Y) \vdash totalSpentFor(Y) |} \\
\text{\{setInitialBalance(X) | changeSetInitialBalance(Y) \vdash setInitialBalance(Y) |} \\
\text{\{fhInterface U fhInstClassInterface U \{super, i+\}}\} \\
\end{align*}
\]

The term fhInstClassInterface denotes the interface of the FinancialHistory environment towards the FinancialHistory class instance, and is defined as

fhInstClassInterface ::= \{newObject, newSubclass, changeReceive, changeSpend, changeGetCashOnHand, changeTotalReceiveFrom, changeTotalSpentFor, changeSetInitialBalance\}

A request handler for metaclass messages has been added to financialHistoryEnv, \( i(m, args) \vdash m(args) \). Most new components simply update the bodies of the methods, but two of them deal with instance creation, newObject, and subclass creation, newSubclass. A detailed explanation for these components will be given in the next section.

The environment of the FinancialHistory class must be also changed accordingly. In order to do this we make the following definitions:

newBody ::= identifier{self} \vdash class(i) \vdash new(r) \vdash i(newObject, r)

newSubclassBody ::= identifier{self} \vdash class(i) \vdash newSubclass(SubClassEnv, SubClassClassEnv) \vdash 
\{newSubclass(SubClassEnv, SubClassClassEnv) \vdash (r \text{ createSub(SubClassClassEnv, i)} | \}

changeReceive Body ::= identifier{self} \vdash class(i) \vdash changeReceive(X) \vdash
Class methods are as before parametrised over the identifier. The body of the class method \texttt{new}, which should create a new instance of the class, first access the name of the internal channel \(i\), used for communication between the class environment and the metaclass, then the return channel, \(r\), which is the unique argument of the message, and subsequently sends the message \texttt{newObject}(r) to the class environment. The body of the class method \texttt{newSubclass} is more complicated, and will be explained in the next section. Other components deal with changing the bodies of the instance methods of the class. Typically, the internal channel \(i\) is accessed by \texttt{class}(i), then the new code of the method by \texttt{changeMethod}(X), where \texttt{Method} is the name of some instance method, and finally the message \texttt{changeMethod}(M) is sent to the class environment.

The definition of \texttt{FinancialHistoryClassEnv} is extended with the following:

\begin{verbatim}
\texttt{financialHistoryClassEnv} ::= 
\{ \ldots \\
| !( \texttt{meta}(m, args) \triangleright m(args)) | \\
| !(\texttt{create Sub}(\texttt{SubclassClassEnv}, i) \triangleright \texttt{internal}(i) \mid \texttt{SubclassClassEnv} \setminus \{\texttt{internal}\} | \\
| \texttt{new wSubclass}(\texttt{new wSubclassBody}) | \\
| !(\texttt{new wSubclass}(X) \mid \texttt{new wSubclass}(r) \triangleright r(X) \mid \texttt{new wSubclass}(X)) | \\
| \texttt{change Receive}(\texttt{change Receive Body}) | \\
| !(\texttt{change Receive}(X) \mid \texttt{change Receive}(r) \triangleright r(X) \mid \texttt{change Receive}(X)) | \\
| \texttt{change Spend}(\texttt{change Spend Body}) | \\
| !(\texttt{change Spend}(X) \mid \texttt{change Spend}(r) \triangleright r(X) \mid \texttt{change Spend}(X)) | \\
| \texttt{change Get Cash On Hand}(\texttt{change Get Cash On Hand Body}) | \\
| !(\texttt{change Get Cash On Hand}(X) \mid \texttt{change Get Cash On Hand}(r) \triangleright r(X) \mid \texttt{change Get Cash On Hand}(X)) | \\
| \ldots 
\}
\end{verbatim}
change TotalReceivedFrom(change TotalReceivedFromBody) 
| ![change TotalReceivedFrom(X) | change TotalReceivedFrom(r) \( \triangleright \) r(X) | change TotalReceivedFrom(X)](change TotalReceivedFromBody)
| ![change TotalSpentFor(change TotalSpentForBody) | ![change TotalSpentFor(X) | change TotalSpentFor(r) \( \triangleright \) r(X) | change TotalSpentFor(X)](change TotalSpentForBody)
| ![change SetInitialBalance(change SetInitialBalanceBody) | change SetInitialBalance(X) | change SetInitialBalance(r) \( \triangleright \) r(X) | change SetInitialBalance(X)](change SetInitialBalanceBody)
| \( \backslash \text{fhClassEnvInterface} \)

The component agent for creating a subclass of a metaclass, `createSub lange SubclassClassEnv,i`, is explained in a following section.

The agent `FinancialHistoryClassInstance` must also be redefined:

\[
\text{financialHistoryClassInstance} ::= \\
\{!\text{class}(c) | \text{identifier}(c) \} \\
\{!\text{class}(m, \text{args}) \mid (v \text{out}(m, r) \mid (r(X) \triangleright X | m(\text{args})) \} \\
\{!\text{super}(m, \text{args}) \mid (v \text{out}(\text{super}, m, r) \mid (r(X) \triangleright X | m(\text{args})) \} \\
\}\backslash \text{fhInstClassInterface} \cup \{\text{super,new,\text{class,identifier},c^+,i^-}\}
\]

However, this is not the whole story, since we must also add reflective capabilities to the environment of the `FinancialHistory` class. This is done by communication, via an internal channel, `(imeta above)` to the `FinancialHistory` class as an instance of the class `MetaClass`. However, we are not going to give an encoding of this class. The relation of this metaclass instance to the `FinancialHistoryClassEnv` is similar to the relation between `FinancialHistoryClassInstance` and `FinancialHistoryEnv`, except that a metaclass cannot create new instances, since it is assumed that there is one and only one instance for each metaclass. New metaclasses and subclasses of a metaclass may nevertheless be created, which must happen via a class message. However, a new subclass must be deployed within the environment of the metaclass, not within the context of the metaclass itself. Therefore we need a new message type, `createSub`, for this purpose. It is intended to be sent from the class to the metaclass asking the latter to start a new subclass. Methods for changing the methods of `FinancialHistoryClassEnv`, mainly `new` and `newSub`, could also be included, but we omit these as well since we are not going to specify the class `MetaClass`.

### 5.5.5 Creating Subclasses

A new object is created within the environment of the class whereof the object is an instance, as was shown in the previous section. The environment of a subclass is also created within this environment. This is done via a message from the metaclass instance that includes a full description of the subclass and metaclass environment, and eventually of any class initialisation procedures. We show here an example with the subclass `deductibleHistory` defined below. This class is a subclass of `FinancialHistory`. Its creation takes place when `FinancialHistory` receives a class message with a request to include a new subclass, e.g. a message `newSubclass(SubclassEnv, Cubclass Class Env)`.

**An Example Subclass: DeductibleHistory**

The example subclass presented below is also taken from [36]. `DeductibleHistory` is a subclass of `FinancialHistory`. Its instances shares the methods of `FinancialHistory`
and adds one instance variable name and four methods for keeping track of the expenditures that are tax deductible. One of these methods, `setInitialBalance`, overrides the same method in the superclass. The class description for `DeductibleHistory` is as follows:

```plaintext
class name          DeductibleHistory
superclass          FinancialHistory
instance variable names deductibleExpenditures
class variable names MinimumDeductions
class methods

class initialisation
    initialize
        MinimumDeductions ← 2300

instance methods

transaction recording
    spendDeductible amount for: reason
        self spend amount for: reason.
        deductibleExpenditures ← deductibleExpenditures + amount

    spend: amount for: reason deducting: deductibleAmount
        self spend: amount for: reason.
        deductibleExpenditures ← deductibleExpenditures + deductibleAmount

inquiries
    isItemizable  
        deductibleExpenditures ≥ MinimumDeductions

    totalDeductions
        deductibleExpenditures

initialisation
    setInitialBalance: amount
        super setInitialBalance: amount.
        deductibleExpenditures ← 0

We make the following definitions:

```plaintext
spendDeductible1 Body ::= identifier self \:
    !(spendDeductible1 (amount, reason) \:
        deductibleExpenses (de) \:
        \(\nu \, (\langle \text{plus}, amount, r \rangle \:\:
            r', \langle \text{plus}, deducing \rangle \:\:
            r', \langle \text{plus}, \text{deducting} \rangle \:\:
            r', \langle \text{plus}, amount \rangle \:
                r', \langle \text{plus}, \text{deducting} \rangle \:\:
                r', \langle \text{plus}, amount \rangle \:
                    r', \text{deductibleExpenses}(de'))\))

| spendDeductible2 Body ::= identifier self \:
    !(spendDeductible2 (amount, reason, deducting) \:
        deductibleExpenses (de) \:
        \(\nu \, (\langle \text{plus}, amount, r \rangle \:\:
            r', \langle \text{plus}, deducting \rangle \:\:
            r', \langle \text{plus}, amount \rangle \:
                r', \text{deductibleExpenses}(de'))\))

|
**isItemizable Body** ::= identifier('self') -> 
  ! (isItemizable | deductibleExp enditures(de) | MinimumDeductions(min) | deductibleExp enditures(de) | MinimumDeductions(min) | 
  (ν r')(de, (reg, min, r') | r'(b) | if [b = true] then r(true) else r(false))

**setInitialBalance Deductible Body** ::= identifier('self') -> 
  ! (setInitialBalance(amount) | deductibleExp enditures(de)) | 
  super(setInitialBalance, amount) | deductibleExp enditures(0)

**deductibleHistoryInstance** ::= setIdentifier(n) -> 
  (identifer(n) | 
  ! (n(m, args) | (ν r)(out(m, r) | (r(X) | (X | m(args))))) | 
  ! (super(m, args) | (ν r)(out(super, m, r) | (r(X) | (X | m(args))))) | 
  (dhInterface ∪ dhState ∪ {super, n +})

dhInterface ::= fhInterface ∪ {spendDeductible1, spendDeductible2, isItemizable, 
  totalDeductions, setInitialBalance}

dhState ::= fhState ∪ {deductibleExp enditures}

dhInstance Interface ::= dhState ∪ fhInterface ∪ dhInterface

dhInst Class Interface ::= fhInst Class Interface ∪ 
  {change SpendDeductible1, change SpendDeductible2, 
  change IsItemizable, change TotalDeductions, change SetInitialBalance}

The deductibleHistory environment is defined as follows:

deductible HistoryEnv ::= 
  (! (i(m, args) | m(args)) | 
  ! (super(m, r) | out(m, r)) | 
  initialise(a) | minimumDeduction(a) | 
  ! (newObject(r) | (ν n) {Deductible HistoryInstance | 
  setIdentifier(n) | 
  (ν r')(m(setInitialBalance, 0, r') | r'(b) | r(n())}) | 
  ! (newSubclass(SubclassEnv, i) | (internal(i) | SubclassEnv) | internal) | 
  spendDeductible1(spendDeductible1 Body) | 
  ! (spendDeductible1(X) | spendDeductible1(r) | r(X) | spendDeductible1(X)) | 
  spendDeductible2(spendDeductible2 Body) | 
  ! (spendDeductible2(X) | spendDeductible2(r) | r(X) | spendDeductible2(X)) | 
  setInitialBalance(setInitialBalance Body) | 
  ! (setInitialBalance(X) | setInitialBalance(r) | r(X) | setInitialBalance(X)) | 
  isItemizable(isItemizable Body) | 
  ! (isItemizable(X) | isItemizable(r) | r(X) | isItemizable(X)) | 
  totalDeductions(totalDeductions Body) | 
  ! (totalDeductions(X) | totalDeductions(r) | r(X) | totalDeductions(X)) | 
  ! (spendDeductible1(X) | change SpendDeductible1(Y) | spendDeductible1(Y)) | 
  ! (spendDeductible2(X) | change SpendDeductible2(Y) | spendDeductible2(Y)) | 

128
The definition of the environment of `DeductibleHistory` class uses the following definitions:

\[
\text{change\,SpendDeductible}/1\,\text{Body} ::= \text{identifier}(\text{self}) \triangleright \text{class}(i) \triangleright \\
\quad \text{change\,SpendDeductible}/1(X) \triangleright \\
\quad \langle\text{change\,SpendDeductible}, X\rangle
\]

\[
\text{change\,SpendDeductible}/2\,\text{Body} ::= \text{identifier}(\text{self}) \triangleright \text{class}(i) \triangleright \\
\quad \text{change\,SpendDeductible}/2(X) \triangleright \\
\quad \langle\text{change\,SpendDeductible}, X\rangle
\]

\[
\text{change\,SetInitialBalance}\,\text{Body} ::= \text{identifier}(\text{self}) \triangleright \text{class}(i) \triangleright \\
\quad \text{change\,SetInitialBalance}(X) \triangleright \\
\quad \langle\text{change\,SetInitialBalance}, X\rangle
\]

\[
\text{change\,IsItemizable}\,\text{Body} ::= \text{identifier}(\text{self}) \triangleright \text{class}(i) \triangleright \\
\quad \text{change\,IsItemizable}(X) \triangleright \\
\quad \langle\text{change\,IsItemizable}, X\rangle
\]

\[
\text{change\,TotalDeductions}\,\text{Body} ::= \text{identifier}(\text{self}) \triangleright \text{class}(i) \triangleright \\
\quad \text{change\,TotalDeductions}(X) \triangleright \\
\quad \langle\text{change\,TotalDeductions}, X\rangle
\]

\[
\text{dh\,Class\,Env\,Interface} ::= \langle\text{dh\,Inst\,Class\,Interface} \cup \{	ext{is\,meta}^+, \text{is\,create\,Sub}, \text{new}, \text{super}\}\rangle
\]

Here is the encoding of the environment of `DeductibleHistory` class:

\[
\text{deductible}\,\text{History}\,\text{Class}\,\text{Env} ::= \\
\quad \langle\text{new}\,(\text{new\,Body})\rangle \\
\quad \langle\text{new}(X) | \text{new}(r) \triangleright r(X) | \text{new}(X)\rangle \\
\quad \langle\text{super}(m,r) \triangleright \text{out}(m,r)\rangle \\
\quad \langle\text{create\,Sub\,(\text{Sub\,class\,Class\,Env})} \triangleright \text{create\,(\text{Sub\,class\,Class\,Env})}\rangle \\
\quad \langle\text{new\,Sub\,class}(\text{new\,Sub\,class\,Body})\rangle \\
\quad \langle\text{new\,Sub\,class}(X) | \text{new\,Sub\,class}(r) \triangleright r(X) | \text{new\,Sub\,class}(X)\rangle \\
\quad \langle\text{change\,SpendDeductible}/1\,(\text{change\,SpendDeductible}/1\,\text{Body})\rangle \\
\quad \langle\text{change\,SpendDeductible}/1(X) | \text{change\,SpendDeductible}/1(r) \triangleright r(X) | \text{change\,SpendDeductible}/1(X)\rangle \\
\quad \langle\text{change\,SpendDeductible}\,(\text{change\,SpendDeductible}\,\text{Body})\rangle \\
\quad \langle\text{change\,SpendDeductible}(X) | \text{change\,SpendDeductible}(r) \triangleright r(X) | \text{change\,SpendDeductible}(X)\rangle \\
\quad \langle\text{change\,IsItemizable}\,(\text{change\,IsItemizable}\,\text{Body})\rangle \\
\quad \langle\text{change\,IsItemizable}(X) | \text{change\,IsItemizable}(r) \triangleright r(X) | \text{change\,IsItemizable}(X)\rangle \\
\quad \langle\text{change\,TotalDeductions}\,(\text{change\,TotalDeductions}\,\text{Body})\rangle \\
\quad \langle\text{change\,TotalDeductions}(X) | \text{change\,TotalDeductions}(r) \triangleright r(X) | \text{change\,TotalDeductions}(X)\rangle
\]
The `deductibleHistoryClassInstance` is defined as follows:

```
deductibleHistoryClassInstance ::= 
  (\{class[\{identifiers[i] \} | identifiers[j]] | identifiers[k] \} | identifiers[l])  
  !(c(m,args) \triangleright (\nu r)(\nu r')(\nu r')(\nu r')(\nu r')(\nu r')(\nu r')(\nu r') \triangleright X \mid m(args))  
  !(super(m,args) \triangleright (\nu r)(\nu r')(\nu r')(\nu r')(\nu r')(\nu r')(\nu r') \triangleright X \mid m(args))  
  )\( \{\{dhInitClassInterface \cup \{super, new, class, identifier, d^+, i^-\}\}\}
```

Creating a New Object

A new instance of class `FinancialHistory` is created by sending the message `new(r)` to the `FinancialHistory` class instance. This results in a new message `i(newObject,r)` intended to the `FinancialHistoryEnv`. Upon receipt of this message, `FinancialHistoryEnv` activates the agent

```
(\{\{\nu n\} FinancialHistoryInstance |  
  setIdentifier(n) |  
  (\nu r')(n(setInitialBalance, 0, r') \triangleright  
  r'() \triangleright r(n))\}
```

The agent `FinancialHistoryInstance` represents an instance of class `FinancialHistory`, abstracted over the identifier name:

```
FinancialHistoryInstance ::= setIdentifier(n) \triangleright  
  (\{identifier(n) \} |  
  !(c(m,args) \triangleright (\nu r)(\nu r)(\nu r)(\nu r)(\nu r)(\nu r)(\nu r) \triangleright X \mid m(args)) |  
  !(super(m,args) \triangleright (\nu r)(\nu r)(\nu r)(\nu r)(\nu r)(\nu r)(\nu r) \triangleright X \mid m(args)) |  
  )\( \{fhInterface \cup fhState \cup \{super, n^+\}\}\}
```

After instantiation of its identifier, a finalisation message is sent to the newly created instance, upon which the identifier name is returned to the initial caller of the action, thus completing the task of object creation. New instances of `deductibleHistory` are created similarly.

Creating the Subclass Deductible History

We shall see how the subclass to `FinancialHistory, deductibleHistory,` may be created. We define

```
dhEnv ::= internal() \triangleright (deductibleHistoryEnv \cup initialise())
```

and

```
dhClassEnv ::= internal() \triangleright deductibleHistoryClassEnv
```

The agents `dhEnv` and `dhClassEnv` may be viewed as the `deductibleHistoryEnv` pending finalisation resp. `deductibleHistoryClassEnv`, both abstracted over the internal channel `i`.  

130
The procedure starts with a message `newSubclass(dhEnv, dhClassEnv)` being sent to `FinancialHistoryClassInstance`. Since the identifier of the latter is `c`, a message `c(newSubclass, dhEnv, dhClassEnv)` is sent. Next the agent `newSubclassBody` is activated within the context of the `FinancialHistory` class. The `newSubclassBody` is defined as

\[
newSubclassBody ::= identifier(set) \triangleright class(i) \triangleright
dCashSubclass(SubclassEnv, SubClassEnv) \triangleright
(\nu \ i') \{ i(newSubclass, SubclassEnv, i') \mid createSub(SubclassClassEnv, i') \}\n\]

The identifier of the `FinancialHistory` class instance is not needed. The internal channel `i` used for communication between the `FinancialHistory` class instance and the `FinancialHistory` environment is accessed first. Thereupon a fresh channel `i'` is created, intended for future communication between `deductibleHistory` class instance and the `deductibleHistory` environment. Finally a message `newSubclass(dhEnv, i')` is sent to the `FinancialHistory` environment, as well as a message `createSub(dhClassEnv, i')` directed to the `FinancialHistory` class environment.

Upon receipt of the message `i(newSubclass, dhEnv, i')` by the `FinancialHistory` environment, the agent

\[(internal(i') \mid SubclassEnv \setminus \{internal\})\]

is activated. This agent simply instantiates the channel `i` in the agent `deductibleHistoryEnv`. The purpose of blocking the channel `internal` is to avoid interference with other eventual request for a new subclass. Is is assumed that this channel is known by the parties, or is globally available.

On the part of the `FinancialHistory` class environment, when the message `createSub(dhClassEnv, i')` has been received, the agent

\[internal(i') \mid dhClassEnv, i' \setminus \{internal\}\]

is activated, upon which the agent `deductibleHistoryClassEnv` is spawned in the appropriate environment.

### 5.6 Conclusions

In this chapter we have introduced a notation, CRCHAM, and have shown how complex mechanisms typical to object-oriented languages may be encoded in the notation. A higher-order version of CRCHAM has been also put forward. Though a series of examples, we have demonstrated how several common data structures may be encoded in the notation. Subclasses may be represented as templates in the first-order version of CRCHAM. A better solution is possible in the higher-order version of the calculus. We have shown how the class and metaclass hierarchies of Smalltalk-80 can be represented in the higher-order CRCHAM by modeling a fragment of the language, the well-known example used in Goldberg [56] to introduce the language. The example is a realistic one, and has been encoded in detail. In general, any class and metaclass hierarchy of the type exemplified by Smalltalk-80 can be expressed in higher-order CRCHAM by the techniques we have presented. Being basically an interactive programming environment, the best way to analyse Smalltalk is by examples, rather than by its syntax.
We have dealt here only with the expressiveness of CRCHAM. We have encoded a series of features common to many object-oriented programming languages which, to our knowledge, nobody has attempted to model. Further investigation is needed in order to develop a semantics in which to reason and prove properties about these constructs. A compositional approach in which the semantics of a system composed of several objects is a function of the semantics of the individual objects would be desirable, but the presence of dynamic binding makes this approach more difficult. We leave this for future investigation.
Chapter 6

Naming and Dynamic Binding

Computer science is an empirical science. Theories of computation have their origin in the way people carry out computation. The Turing machine is a distillation of computational behaviour, of the way people compute with paper and pencil. The nature of computation is well understood today, and has been rigorously described in different models which turned out to be equivalent, e.g. Turing machines, lambda calculus, Post machines, context free languages.

The same cannot be said about about the nature of communication and interaction. These are no models for interactional behaviour that rigorously describes communication, whose nature is still an open question. In this thesis we examine the nature of names and dynamic binding of names. However, this question is intimately connected to the one about communication and interaction in general. And since this is still an open question, the best way to carry out this inquiry, in our opinion, is by turning our attention to the way in which existing systems and formalisms treat communication and interaction. Our purpose, nevertheless, is not to formulate a general model of interaction, only to look into the way that names and binding of names appear in those systems and formalisms. We believe that this inquiry may throw some light on the more general question concerning communication and interaction.

In this chapter we show that in any notation intended for the formal description of open systems, either some form of dynamic binding of names is present, or its absence gives rise to awkward descriptions. We argue also in favour of the adequacy of the object paradigm and operational semantics to describe systems of concurrent objects. Since we regard computer science in pragmatic terms as an empirical science, we see the results obtained from this kind of study as the best evidence supporting our claim that dynamic binding of names is a necessary feature in any open system of communicating agents of some complexity.

Although the material presented below is extensive, no attempts have been made at being exhaustive. This chapter is not intended as an overview of the literature on the subject, but rather as part of an inquiry into the nature of naming and dynamic binding based on the evidence offered by the way these phenomena appear in existing systems and notations, and as such it should be considered as an integral part of our research on the topic. The presentation of the distinct formalism and languages is thus interwoven with commentaries and observations that lend evidence to the major claims of this thesis. Accordingly, the criteria for the selection of the material have been the relevance to the topic under investigation, and we have focused
our attention mainly on those works which we believe are most representative of a
certain line of research and that have been more influential.

The rest of this chapter is organised as follows. In Section 6.1 we concentrate on
the semantics of names. There are sections dedicated to the subjects of pure names,
nominal calculi, and the object paradigm. Section 6.2 is dedicated completely to the
presentation and discussion of the functional paradigm. Previous research including
the presentation of several notations is presented and discussed, with special focus
on the role of names in these notations. In every relevant case we emphasise what
we believe are the limitations of this approach in giving a meaning to concurrent
objects in open systems. Finally, in Section 6.3 we present the object paradigm,
which we also call the concurrent paradigm. The fact that this section is much
shorter than the former attests to the hegemony of the functional paradigm at least
until very recently. In this section we present the notions of actors and actor-based
systems, transducers, and several attempts to encode objects in process algebra
languages. We discuss also the limitations imposed by the absence of primitives for
encoding dynamic binding

6.1 The Semantics of Names

In this section we focus our attention on the notion name and its semantics and
consider some works were the the notion of name has been explicitly considered.

Essential to the notion of name is the fact that names may be used to define more
complex structures or expressions, and that the equality relation may be seen as a
matching relation between distinct occurrences of names, or tokens. In this sense
the equality relation refers to the elements of the language itself, i.e. the tokens, not
to the the terms of the metalevel language, as in the \(\lambda\)-calculus. In the \(\lambda\)-calculus,
an assertion of equality between variables, \(x = y\), can be regarded as a meta-level
assertion about the language used to describe the calculus, rather than about the
calculus itself. The concepts of name, token, token matching and expression are
enough to express and manipulate data structures of any desired complexity. A
theory of computation might be defined solely in terms of manipulations of these
structures based on the operation of matching.

6.1.1 Pure Names

Needham [123] defines a pure name as “nothing but a bit pattern that is an identifier,
and is only useful for comparing for identity with other bit patterns - which include
looking up in tables in order to find other information.” In the following, we will
refer to pure names simply as names.

Names may be used for many other purposes. In denotational semantics they con-
stitute the domain of functions that maps identifiers to abstract values. In process
calculi they may denote channels or agents. But the most important feature is
probably that they may be used in references. Names can be compared with each
other for equality, and may thus be used for looking up tables “in order to find other
information”, a crucial feature. Many dynamic features in programming languages,
including dynamic binding, depend on this use of names.

It is not surprising that pure names have been studied mainly in the context of
distributed or modular programming. Whenever we have to do with systems where
communication among distinct elements is necessary, e.g. in incremental program
environments, program modularisation, distributed systems, message passing in
object-oriented systems, or distributed database systems, the notion of pure names

134
seems to play an essential role: “names are frequently used for communication about objects” [123]. In purely sequential languages names may always be alpha-converted, and even compiled away. This is not true where some form of communication among independent agents is necessary.

In distributed systems there is “more naming because there is more binding.” Machines must be bound to addresses, services to machines, etc., “neither of which has a very obvious analogue in a centralised system.” This fact is largely a result of the need for communication between distinct parties. Machines are bound to addresses in order that they may be made available to other machines, and services are bound to machines in order that they may play the role of service providers to clients. Dynamic features are essential in such systems, as exemplified by Needham’s words that “it is the conventional wisdom of distributed programming that in any cases of this sort early binding is extremely wicked, and every opportunity must be taken to allow for variability.” Variability requires some form of late or dynamic binding, whereas early binding is closely related to the notion of a fixed static binding.

The binding of a name need not be unique, and the interpretation of names may be complex. Even in systems where name binding is assumed to be unique, the question of where to look up for a binding may arise because the lookup tables are often replicated for efficiency reasons, and a consistency problem may arise as a consequence of table updates. In systems where names are not assumed to be unique, an environment must be associated with any name occurrence. This might be by default the current environment, e.g., in shallow binding, common in languages with dynamic binding. By contrast, in static scoped languages an environment might be attached to a name, cf. the notion of a function’s closure. Furthermore, environments may be nested, and the lookup operation may involve searching for a name binding in several environments.

In general, the semantic of bindings varies widely among different languages. For a short account, see Huyens and Quennec [131].

6.1.2 Nominal Calculi

In Gordon [57], Andrew Gordon argues that pure names are relevant for mobility. He defines a nominal calculus as “a computational formalism that includes a set of pure names and allows the dynamic generation of fresh, unguessable names.” He stresses the fact that both the \( \lambda \)-calculus and “a variety of imperative, object-oriented and concurrent programming may be reduced to the \( \pi \)-calculus,” in which channels are represented by names.

Names may be used for other purposes in process calculi, e.g., for identifying locations. Amadio and Prasad [15] propose a new nominal calculus, the \( \pi \)-L calculus, in which processes and channels may have locations that are identified by names. Examples of nominal calculi without locations in which processes may be passed as values are CHOCS [169] and the higher-order \( \pi \)-calculus [149]. The choice free asynchronous \( \pi \)-calculus, on the other hand, dilutes the identification of names with channels in the \( \pi \)-calculus. This separation of names and channels is conspicuous in e.g., the reflexive chemical abstract machine [48], in which channel-based communication is dropped in favour of other primitives. In Mobile Ambients [32] names refer to ambients, a cluster of processes which moves as a group. Nominal calculi and the capability of producing new, fresh names, have also been applied for modelling notions of security in cryptographic protocols. In the spi calculus [4] names are used as cryptographic channels representing cryptographic keys.
The Properties of Names

The functionality provided by the notion of name and name space became manifest by the time that the need was felt for imposing structure on memory systems and storage allocation. Holt [185] suggests breaking up a large total program into allocation units, which are allocated separately to memory whenever the computation requires them. This structuring of allocated memory required a flexible referencing system. Iliffe and Jodeiot [71] proposed the use of indirect addressing though code-words as a way of controlling bindings. The design of Multiprogramming systems was the subject of Dennis [20], who contrasted the concept of name space, i.e. the set of addresses a process can generate, to that of memory space, the set of physical memory locations, and memory referencing schemes described by address mappings from name space into memory space. The association mechanism name/location map was proposed to solve the allocation problem for main memory. Since a data object needed to be loaded several times at different addresses, an extra level of indirection would be required to relate addresses used by the computation to the physical locations. The functionality provided by names, name spaces and table lookup became essential to solve this problem, and the role of names in sharing and modularity came into the open.

Fraser [11] did a pioneering work in the study of the relation between naming in languages and naming in systems, and of contexts and manipulation of contexts. He was to our knowledge the first researcher that proposed an extension of the λ-calculus to deal with the notion of context (see Section 6.2.6 for more details). Fraser recognised the role of names to identify items of data, where each item occupies a storage location, and to identify contexts. Names themselves must appear in some context, and the result was a hierarchy of contexts.

The most accomplished early treatment of the property of names in computer systems was given by Saltzer [146]. Saltzer distinguishes at least three different uses of names. The first concerns naming of individual variables, the second naming in database management systems. The third one, “somewhat less systematically studied”, concerns concerns the “collection together of independently constructed programs and data structures to form subsystems, inclusion of one subsystem as a component of another, and use of individual programs, data structures, and other subsystems from public and semi-public libraries.” Saltzer points out that “such activity is an important aspect of any programming project that builds on previous work or requires more than one programmer... In this activity, a systematic method of naming objects so that they may contain references to one another is essential... If true modularity is to be achieved it is essential that it be possible to refer to another object knowing only its interface characteristics.” This characterisation of the use of names is very relevant for the purposes of this study. It emphasises what we believe is the most important function played by names in modular or distributed systems: that of communication bearers. Whenever distinct parts of a program have to communicate with each other or be integrated in some other fashion, e.g. in incremental program environments or modular programming, the function provided by names seems to be essential. As Saltzer pointed out “the need for systematic approaches to object naming has only recently been appreciated” as a consequence of “the arrival on the scene of systems with extensive user-contributed libraries and the potential ability easily to ‘plug together’ programs and data structures of distinct origin.”

Another important observation made by Saltzer is that “the use of an object should not mean that the user of that object is thereafter constrained in the choices of names for other, unrelated objects.” But although this goal is “obvious”, it is nevertheless “surprisingly difficult to obtain, and requires a systematic approach to
naming.” In our view this systematic approach concerns not only the capability to create fresh names, but also to specialise the use of names locally, e.g., by dynamically binding a name in a determined local domain and in this way give it a special, context-dependent meaning within this specific environment. Saltzer points out that a “systematic semantics has not yet been developed,” an assertion which we believe it is to some extent is still valid today. To our knowledge, the first conscious attempt to develop a semantic of names seems to to be Milner [111]. However, the use of names in process calculi was apparently not a conscious choice, which may explain why the phenomenon of environment and dynamic name binding was not given due attention in these formalisms. The result is that there has been lately a movement towards formalisms supporting the notion of environment, context or ambient, i.e., the “universe of discourse” and dynamic binding in the form of local communication. This is the case of such formalisms as LLinda, the Seal Calculus, the Ambient Calculus, and others, which were presented in Chapter 2.

Saltzer clearly approaches names from an object-oriented point of view. His approach is probably similar to the one taken by Milner when he contrasted what he called the object paradigm against the functional paradigm [110] (see next section). According to Saltzer, objects may be any computation entities, e.g., array of bits or other more structured entities that may even contains other objects. This containment may be by value, i.e., a copy of the component object may be included in the containing object, or by name, where only the name of the object, not the object itself, is included in the containing object. The former is common in the functional approach, where the notion of value is essential but not that of entities with state, whereas the latter is akin to the notion of containment in object-oriented languages. The scheme of containment by value in object-oriented languages, according to Saltzer, is inadequate “because it does not permit two objects to share a component object whose value changes”, an essential feature in object-oriented systems, where objects are supposed to be persistent and stateful.

Apart from sharing, the purpose of naming is thus to allow inclusion by name of component object within a containing object. This is a particular case of the fact that names may be associated with constructs that may change over time and thus require a lookup operation whenever it is accessed, a powerful dynamic operation which can scarcely be “compiled away”. As Saltzer points out, “when names are used, some way is needed to associate the names with particular objects... It is common for several names to be associated with the same object, and for one name to be associated with several objects for different purposes.” This is accomplished by one abstract pattern into which these possibilities fit, which Saltzer calls a context: “a context is a partial mapping from some names into some objects of the system.”

When the program interpreter has to refer to an object, “it accomplishes this reference by looking up the name in the associated context.” Binding a name in a context is the act of “arranging that a context shall map a name into an object”, and resolving a name in a context is “using that context to locate an object.” Another important concept is closure, which is “fundamental to naming”, according to Saltzer, specially in connection with “the problem of changing contexts”. Saltzer define a closure as “the mechanism that connects an object that refer to other objects by name with the context in which those names are bound.”

As observed by Saltzer, in the presence of sharing the goal of modularity is not accomplished if “the use of shared object requires that the user know about the name of the objects that the shared object uses”, for example “by avoiding the use of those names.” This observation suggests that agents must either be endowed with the capability either to create fresh names, a very difficult task, or else be able to construct separate name spaces or contexts in order to avoid possible name
clashes.

Contexts are also required in the case of unstable bindings, i.e., bindings that may change unpredictably between definition and use. The typical case is file system catalogues, where names may be deleted, added or rebound. Nevertheless, this case may also include the mutable bindings of variables in imperative sequential languages. Mutable variables are often seen as a mathematical aberration, but they seem to be the most common and natural element in computer languages.

Names with different properties are necessary. A single object “may have many kinds of names, appearing in different contexts.” An object might need a distinct, unique identity in a system with multiple users. This may be accomplished by “some form of universal name, resolvable in some universal context.” Notwithstanding, an individual user or program may need “to be able to refer to objects of current interest” with local names, without knowledge of the universal names. Moreover, “one may need to have synonyms.” Thus, the “unique name property” is not always desirable.

The notion of communication can also be applied to the case in which several programmers are developing separate functions that are meant to be linked together, and need to choose names both for private temporary variables and for communication among the different functions. As noted by Saltzer, most language systems “have been designed to aid the single programmer in creating programs in isolation... It is only secondarily that they have been concerned with interaction among programmers in the creation of programs.” In some sense this problem seems to linger on to this day.

The reason why systems requiring complex naming policies were nevertheless already a reality by the time of Saltzer’s article, is that, as he pointed out, “communities tend to adopt protocols and conventions for system usage that help programmers to avoid problem.” Furthermore, since the use of file systems is meant to be used by humans, ambiguity can be often resolved “by asking questions.” This type of ad hoc approach to naming is apparently still prevalent in current systems.

The question concerning name binding in distributed systems was also raised by Saltzer. He points out that “there are several relatively interesting naming topics about which noting systematically is known, and the few case studies in existence are more intriguing for their irregularity, inconsistency, and misbehaviour than for guidance on naming structure... These topics arise whenever distributed systems are encountered.”

6.1.4 Milner’s Object Paradigm

In a paper dedicated to the study of functions as processes [110] Milner introduces what he calls the object paradigm, he associates with the “links-as-values” approach of the π-calculus, in contrast to what he calls the functional paradigm. The basic notion in this paradigm is the transmission of access to an object, rather than transmission of the object itself, cf. the notion of containment by name in the previous section. According to Milner “both paradigms are equally basic and significant.” He considers that “some translation from the function to the object paradigm must exist, since functional languages are successfully implemented on machines which - with their arithmetic units, programs and registers - can be see as assemblies of objects.”

Denotational semantics commonly translates imperative features into something which can be viewed as another kind of language, with continuations for sequencing, function calls for assignment, etc. This fact received a thorough treatment in
Steele [161], which we discuss in Section 6.2. It is called the continuation-passing style of programming, and may be seen in some sense as a kind of implementation of the denotational semantics approach. It incorporates imperative features in a functional form. On the other, Milner has shown [110] that functional language may be encoded into the $\pi$-calculus. Both approaches seem to thus be equivalent. Nevertheless, the object paradigm seems to be more general, since it is adequate for describing systems that are not meant to produce any result, i.e. reactive systems. It is well known that the functional paradigm is not suitable for describing parallel, distributed or open systems, which are more naturally described as a composition of agents or active objects. We will return to this issue in section 6.2.3, where we discuss an attempt at giving a denotational semantics to an object-oriented language. This semantics incorporates features which are common in the object paradigm, but rather awkward in the context of denotational semantics, since unconventional domains must be defined that include syntactic name domains in position other than in the domain of semantic functions, and resorts to lookup operations to resolve the meaning of such names. In other words, it incorporates constructs that are natural in the object paradigm, thus corroborating our view that the latter is more basic. Furthermore, there is a problem with the use of fixed-point operations, mandatory in denotational semantics, since this procedure in some sense “freezes” the meaning of constructs which should be regarded as basically undetermined, dynamic or “open” (see Section 6.2.3 for a discussion of this point). Thus, new subclasses cannot be introduced without changing the denotational meaning of programs. This problem is absent in the object paradigm and its corresponding operational semantics, which is usually based on transition systems labelled by what may be regarded as the free names of agents, i.e. the barbs or means of communication of an agent that is open to the outside. The operational semantics captures this feature adequately.

These facts suggest, in our opinion, that it would be more adequate to adopt the object paradigm for giving semantics to object-oriented languages, which are concurrent in nature. This has also been done lately, but the basis for such posture has not been explicit in a detailed and consistent manner to our knowledge. What is missing in the link-as-values paradigm, in our opinion, is support for the notion of context, context switch, and dynamic binding of name within contexts, which requires some form of higher-order communication as in the functional paradigm.

Milner noted also that that concurrent computation and “the power of concurrent active agents to influence each other's activity on the fly cannot be forced into the 'function' mould... without severe distortion” [111]. His conclusion in this paper, which is more radical than in Milner [110], is that “the functional computation is a special case of concurrent computation”, an insight we try to expand in latter sections. Milner proposes the use of naming as the pervasive notion in models of concurrency, among other reasons because “the act of naming, or address, is inextricably confused with the act of communication.” We have tracked this use of name back into the transducer model [110], which we analyse in Section 6.3.1. We have found that names are present in every communicating system we have encountered. Another reason for distinguishing naming, according to Milner, is that “naming strongly presupposes independence: one naturally assumes that the namer and the named are co-existing (concurrent) entities.”

Although the influence of Milner has been large, it seems that his insights have not yet been given an adequate treatment. One exception might be the work of Laurent Dami [41], although he treats this fact only superficially (see Section 6.2.5). We should maybe also mention the work on active objects of Nierstrasz in this context [127], although the issue of naming does not seem to have been explicitly targeted there. David Walker [179] worked on giving formal semantics to simple object-
oriented systems along the lines proposed by Milner.

But a fact remains problematic. As Milner points out “a challenging problem is to reconcile the assumption, quite common in the world of object-oriented programming, that each object should possess a unique name with the view... that naming of channels, but not of agents, should be primitive in the π-calculus.” In fact, the asynchronous version of the π-calculus dilutes the π-calculus association of names with channels, and approximates it to the Linda calculus or the Chemical Abstract Machine in the sense that the communication media consists of an undetermined solution where messages may be dropped and picked by matching or made to react according to any reaction rule we may define. Paraphrasing Needham [112], we could say that “names commit us to nothing.” In other words, we can use names in any way we desire, the only “ontological commitment” being that they can be matched with each other for equality and are pure, i.e. have no intrinsic structure.

6.2 The Functional Paradigm

Objects have been widely examined in a functional setting, whose theoretical foundations are logic and the lambda calculus. Programs are viewed as a functions from a domain of input values to a domain of output values. Computation evolves via “side-effects”, i.e. commands transforming the bindings of a certain global environment or store.

This rather procedural approach towards the semantics of computer languages is implicit in the continuation-passing style of the denotational semantics, ans also of functional programming as proposed by Steele [161], where Scheme’s lambda expressions are used not as functions, but as control structures. The fact that a functional programming language is used as an imperative language is less surprising in light of the fact that the denotational semantics of imperative languages in continuation-passing style may be viewed as a form of functional language with continuations mapping stores to stores. Since functional languages such as Scheme are also implemented in Von Neumann machines, in some sense their interpretation must be representable in an imperative style. Both paradigms are thus equivalent and interchangeable. However, we believe that the imperative or procedural approach is the primary one in computation, which is not always meant to yield a result, cf. reactive systems. Turing machines may also be described as a finite set of commands that manipulate the environment, represented by the potentially infinite tape set.

There are languages in which stores are first-class citizens and can be assigned as values of variables. Thus, in Johnson [76], a programming language, GL, is presented in which stores and continuations are first-class objects, and a semantics is given in which “stores are functions from identifiers to integers” and “an entire program produces a store as its only result; there is no input or output.” As pointed out in Steele [161], “it is possible to write useful programs in terms of LAMBDA expressions in which we never care about the value of any expression.” In other words, the functional continuation-passing programming style may be viewed as a disguised form of imperative programming in which the imperative features are concealed by the interpreter. By analogy, the denotational semantics of programs may be viewed as a translation from one language into another that conceals the imperative features. The difference is that there is no assignments and thus referential transparency is preserved. But as we shall see it is not always possible to rule out assignments, which have been shown to add to the expressiveness of a purely functional language.
The denotational description of programming languages, as we have seen, often includes features which are naturally associated with the imperative programming paradigm, not with the functional paradigm. This seems to contradict the alleged abstractness of such descriptions, and strengthens the thesis that denotational semantics translates programs to another language which basically tries to mimic the behaviour of imperative constructs in a rather unconventional functional framework.

The rest of this section is organised as follows. In Section 6.2.1 we contrast the notions of names and variables in the context of recursion, recursive definitions, and self-reference. Section 6.2.2 is dedicated to the topic of environments and closures in a functional setting. Section 6.2.3 concerns the denotational semantics of objects as closures, whereas Section 6.2.4 is dedicated to the presentation and discussion of a proposed denotational semantics for concurrent objects. In Section 6.2.5 we present a study that is very relevant for the subject of this thesis, in which names and dynamic binding are explicitly dealt with in an attempt to integrate functional and object-oriented features. The last section, Section 6.2.6, is intended as both a survey and a discussion of several formalisms, languages and systems that we found specially relevant for our subject.

6.2.1 Recursion and Self-Reference

The use of fixed points to model references to self in objects appeared for the first time in Cardelli [31]. In this interpretation, objects are records, possibly with functional components, message passing is field selection of functional record components, and inheritance is an extension of the set of record fields of a determined record. A record is a data structure with a fixed number of fields, i.e., components named by labels. Labels are considered to be an essential part of records.

Records are finite associations of labels and values, which in the case of objects consists of functions over messages. In order to allow a record field to refer to other components of the same record, objects are endowed with a self-referential capability by the introduction of the rec operator, used to define recursive functions and data. Thus, an object may be defined according to the following pattern:

\[ \text{rec self } [l_1 : \lambda E_1, \ldots, l_n : \lambda E_n] \]

In this construct, self may occur in any of the expressions \( E_i \).

It has often been claimed that objects are by nature recursive, but the meaning of this assertion is not clear. This statement seems to ascribe to objects what is rather a feature of the functional interpretation of objects rather than a feature of the objects themselves.

We can compare this assertion to a similar one about functions. What would it mean to say that a function is recursive? A function may certainly be defined recursively, and it may also belong to the collection of \( \mu \)-recursive functions, but this does not make it recursive "by nature" without further qualification. This is even more apparent, in our opinion, when the epithet is applied to the notion of object. An object may be seen both as a kind of data structure and as an active computational agent that manipulates these data structures. We believe it is more adequate to apply the notion of recursion only in relation to computational agents, not to passive data structures. In consequence, we regard the pseudo-variable self in object-oriented languages as any other kind of reference. There is as such nothing recursive in the fact that the contents of a memory cell in a computer memory may be the address of the cell itself. To us this is just any address, and the fact that it turns out to be the address of the cell containing that address is a contingency that
may be interpreted in several ways. It might involve some kind of circularity or loop, depending on how it is interpreted, but the circularity is ultimately a feature of the interpreter, not of the data. In fact, loops are not strictly necessary to encode what is supposed to be a recursive structure, as attested by the notion of replication in π-calculus. Hence, only an active computational agent should be classified as recursive by nature, at least when it is coded in a finite-control structure but is capable of a potentially infinite behaviour. In this case, since the data representing control is finite, configurations in the control structure must necessarily recur. Now, it is true that with the stored program concept even the control structure may be seen as data. In fact, the universal Turing machine already involves the notion of stored program, at least implicitly. The computational agent of a universal machine executes recursively the commands defined by the program stored in a tape or some other piece of memory. Nevertheless, the computational agent itself is never formalizable within the language itself that it is supposed to interpret or act upon. Likewise, in the λ-calculus and in functional languages recursiveness may be seen as a feature of the computational agent, which in this case is the interpreter or metacircular evaluator that repeatedly performs the evaluation-application loop.

If self-reference in objects is regarded as any other reference, as it is implicitly done by the object paradigm (described in Section 6.1.4 below), the fixed-point semantics which usually follows may be avoided. The constructs obtained by applying fixed-point techniques at the level of an object’s self-reference are usually uninformative, in contrast to what is the case when it is applied to recursively defined functions. On the other hand, the behaviour of an object may be classified as recursive, in the sense that an object is usually ready to continuously receive messages from other objects and apply them, e.g. by invoking a method referenced in the message. This property is encoded by replication in the language CRCHAM described in Chapter 5. The methods of an object, on the other hand, may certainly have been defined recursively, but the use of self-reference at this point is not mandatory, since a method invocation may be considered simply as an internal or local call.

The distinction between recursion and self-reference is analogous to the one between names and variables, two concepts which we would like to keep separate. In a functional setting variables denote values, and identifiers are often defined as a textual name that refers to a variable. Pure λ-calculus can even be described as a name-free calculus. Variables in the λ-calculus are mere place-holders, and the identifier used to represent it may be substituted by any other, as far as the substitution function is injective. The main difference between names and variables is that names may be used point at objects or expressions, whereas variables denote abstract values. In our interpretation, names are syntactic (textual) categories that do not simply refer to variables, nor denote values. They may be used to point at values, but are not instantiated by these. In contrast to variables, the relation between a name and the entity it represents is dynamic, i.e. it may change over time. This seems to defeat any attempt at giving a fixed-point semantics to names associated with terms in which it may occur. This association should not be viewed as a recursive definition, but only as an association between terms in some syntactical category. Dynamic binding highlights this difference between denoting a value and referencing a term, since here the term referenced by an unbound identifier depends dynamically on the context in which it may occur.

In the pure λ-calculus a value is a closed term. Since the variables are mere place-holders, the calculus can also be expressed without variables, with combinators or de Bruijn indexes. Nevertheless, the use of names or free variables is a very powerful computational idea. Although theoretically feasible, in practice it would be quite cumbersome and inefficient to avoid the use of names.
In functional programming, free names may be used for naming functions, and may also appear as undefined variables in expressions which are intended to be bound statically, e.g. in an incremental programming environment, or as free variables to be bound dynamically in the context of application. When variables are bound by a $\lambda$, it would be possible during function application to substitute them immediately in the body of the function being applied, but this would be very inefficient. Although substitutions as such do not belong to the $\lambda$-calculus, their importance from the computational point of view have induced researchers to treat them as part of the logic, e.g. in the $\lambda\sigma$ calculus [2].

Names are used for naming functions, an essential feature in e.g. incremental programming in an evaluative context, as well as for the definition of recursive functions. Since it was soon discovered that $\lambda$-notation is inadequate for naming functions defined recursively, McCarthy introduced labels, in other words names, to denote expressions [100]. The expression label$(a, \varepsilon)$ was proposed to denote the expression $\varepsilon$ in which occurrences of $a$ are interpreted as referring to the expression as a whole. Denotationally, this could be defined as $\mu a. \varepsilon$. The symbol $a$ in label$(a, \varepsilon)$ can be regarded as bound.

There are two ways to express recursion: binding the function to its own name, as in let(rec)-expressions or list-expressions in LISP, or with the help of the "paradoxical combinator", the Y-operator.

In the latter case, recursive procedures may be defined without recursion. In principle, this operator can be used to implement recursive local procedures, but it is extremely awkward to use it for the construction of mutually recursive procedures, to the point of significantly obscuring the semantics of the language. The LISP compiler, for instance, produces much worse code for functions defined with the Y-operator than when using assignment. In the later case a run time closure for the lambda expression is constructed, whereas in the former the generated code branches to the entry point on each recursive call [145].

Recursive procedures are usually defined with the help of let(rec)-expressions, which involve the use of names and dynamic lookup. In principle, at least, this may be considered as an embellishment of name-free Y-expressions. For instance, a let-expression

$$\text{let } f = E \text{ in } E'$$

could be easily expressed as a lambda abstraction

$$(\lambda f. E')E.$$

One problem, though, is that in the first case $f$ may be applied several times to many different arguments within $E$. Similarly, in the case of incremental programming, a define-expression

$$\text{define } f = E \text{ in } E'$$

may be defined at the top level and then used in any part of the program. Even letrec-expressions

$$\text{letrec } f = E \text{ in } E'$$

where $f$ may occur in the expression $E$, are easy to formulate in terms of the Y-operator.
\[
\text{letrec } f = E \text{ in } E' \equiv (\lambda E')(Y(\lambda f.E))
\]

This transformation poses many problems, however, e.g. programs may be harder to type check and evaluation of expressions might become less efficient. Furthermore, expressions involving several mutually recursive function definitions are not so easily transformed into name-free lambda calculus expressions, and questions may even arise concerning the faithfulness of the translation.

In any case, the simultaneous presence of assignment and \textit{letrec}-expressions seems to defeat any sensible translation into the \(\lambda\)-calculus. Steele and Sussman [161] (see details below in this section) show how several programming constructs, including assignment, are expressible in an applicative notation involving only local translations that preserves behaviour. Nevertheless, as pointed out by Felleisen [45], the translation of control statements and assignments involve global reformulations of programs. In fact, according to a formal notion of expressiveness presented in this work, it is shown that the Pure Scheme cannot express assignment and \textit{letrec} constructs together. The counterexample is

\[
\text{letrec } f = \lambda x.(\text{set}! f \lambda x.\Omega)
\]

where \(\Omega\) denotes non-termination, and \text{set}! a destructive assignment operation. The first application of \(f\) converges, but a second invocation leads to divergence since the procedure has been now modified. Thus, assignments increase the expressive power of Pure Scheme. This example also illustrates the the limitations in expressiveness associated with a computational model that excludes assignments for the sake of referential transparency.

In assembly language names are ubiquitous as register names, memory location, etc. By contrast, names are absent in data flow languages, where it is possible to draw arrows instead. But it is in connection with the use of free variables in languages with dynamic scoping rules that names really come into their own. Names occurring free in an expression are not \(\alpha\)-convertible, and the expression itself must remain unevaluated. Two terms that differ only by their dynamic variable names are not equivalent. It is in this context that the distinction between names and variables is most evident.

\textit{6.2.2 Environments and Closures}

An environment is a structure that provides contextual information required for finding the value of an expression, and is a pervasive notion both in programming languages and denotational descriptions of programming languages. The aim of denotational semantics is to give a correspondence between programs and mathematical entities. Denotational descriptions use identifiers or names, a syntactic category, as domains of environments. Environments are what bind together programs and the semantic domains that give programs a denotational meaning, and are defined as functions mapping a pure syntactic domain of identifiers to a pure semantic domain of values. In functional languages environments consist of mappings from identifiers to \textit{expressions}, which in contrast to values in a semantic domain are symbolic. This shows the close relationship between the concrete data structures used for the evaluation of programs in interpreted functional programs and the abstract mathematical entities giving denotational meaning to programs. As noted in [105], “if we are trying to explain what happens when a program is executed we need to keep track of how the values associated with names change as the execution
of a program proceeds.” This is done by “using the fact that at any point during the execution the association between values and names can be regarded as a list or function”, i.e. as an “environment”. The denotational definition of programs follows closely the same pattern: “for us environments are abstract mathematical entities which simply happen to model portions of machine states.” We could thus question the “abstractness” of this kind of entities and the fact that they “simply happen” to model machine states [105]. In fact, their relation to program states and program execution is much less casual than suggested by these words. As the same author points out, “the execution of a program is initiated by applying a valuation to the program” (my italics), and “it can be helpful to note that valuations are applied in the order in which machine instructions are obeyed.” It seems we are executing the problem, but in another, closely related, language. Paraphrasing these words, if we substitute “symbolic expressions” for “values”, we could say that what happens when a program is executed” is exactly that the “values associated with names change” as a result of the execution of a program, and that this is exactly what a program does when it executes.

The example usually given in support of a mathematical semantics for languages is the contrast between numerals and numbers, where the numbers are mathematical abstract objects, and the numerals are expressions in a certain language (binary, octal, roman decimal, etc.) that “should not be confused with the concepts they denote” [155]. The analogy is faulty because numbers seem to be the only mathematical object whose intuitive meaning nobody denies, as exemplified by Kronecker’s famous remark that “God created the integers, all else is the work of man”, or by recent neurophysiological theories claiming that the number sense is wired in to our brains at birth [65]. We can scarcely say the same of computational constructs for modern Neumann machines.

This raises the question whether concrete languages are not the only thing “that exist” [60]. As pointed out in [60], it is possible to use the notation of denotational semantics as a metalanguage, denying that it refers to abstract entities. Usually this approach is dismissed without much argument, although there is “some well-developed mathematics there to build on” [162]. According to Stoy “people who succumb [to this approach] can frequently be recognised by their excessive preoccupation with the actual symbols as written”, hardly an argument.

Pure functional languages (without assignment) are based on the principle that identifiers and variables are names for values, and that each one of them may refer to one and only one value. A computational model for a functional language must provide this association of identifiers and values. Usually this is done by the introduction of data structures called frames or environments, commonly a list of pairs, each pair consisting of a name and a value in some domain. Let us illustrate this by showing how a definition of a function $f$

$$f ::= (\lambda x)\varphi[x]$$

might be evaluated by e.g. a Scheme interpreter: Here $\varphi[x]$, the body of the function $f$, is some wff of the language, possibly containing some undefined identifiers. The result of the evaluation is an environment where $f$ is bound to the the procedure object, which consists of a parameter list containing $x$ and the body $\varphi[x]$, as well as a pointer back to the environment. The body is represented textually. We observe thus the presence of a self-reference or loop. The environment holds a pointer to the procedure object, and the latter holds another pointer back to the environment. This could induce one to believe that the resulting structure is essentially recursive. But the recursion involved here is deceptive. If we examine the evaluation process of the Scheme interpreter, what we really get is a frame structure consisting of simple
pairs relating an identifiers and a value, which in the case of \( f \) above is a procedure object, consisting of a list of arguments and the text of the body of the function. In other words, what we have is simply

\[
\rho := \{ f : ([x], \phi), \ldots \}
\]

without any recursion. The text of the function body, \( \phi \), may contain unbound identifiers that have not been declared yet. If \( f \) is evaluated with some actual parameter for the variable \( x \) with unbound identifiers in \( \phi \), an error may occur. But this will happen only at runtime, i.e., interpretation time, i.e., during evaluation of a function application, not at declaration time. In a certain sense we may thus talk here of a kind of late binding. Thus, identifiers occurring in \( \phi \) are left undefined until their values are required. This relies on the fact that the value of a lambda expression does not require evaluation of its body until the function is called. It is implicit that those values are looked up in \( \rho \), which is the only environment involved in this simple example. Accordingly, the self reference noted above is superfluous. In fact, there is a trade-off between the use of self reference in the data structure presented above as an aid to the evaluation process, and a dynamic operation consisting in looking up the contents of the environment, executed at runtime by the interpreter.

In the presence of locally defined functions (for instance by \texttt{let} constructs), and even during function application to instantiated variables, a more complex environment structure is necessary, and several environments may exist at the same time. In this case, references to environments must be explicit. The evaluation process must now be given an environment as argument, since every expression must be evaluated in some specific environment. Self reference is used here for reasons of efficiency, but it not strictly necessary. Roughly, this is done by attaching environments dynamically to the body at interpretation or run time. We omit the details. What we want to stress is the fact that self references in the data structures used to control the flow of execution may be substituted by a more dynamic “runtime” behaviour at the cost of extra lookups at tables or the need to pass around extra values or pointers during execution/interpretation.

Another possibility is to let the machine representation of a function consist of two components [162]: the code of the algorithm, as before, and a free variable list, rather than an environment, giving the values denoted by the non-local variables. Thus, only the variables actually mentioned in the body of the function are accessible. Once again, we have a closure whose structure does not require the presence of self-references.

If we turn to the stack implementation of a functional language, in which expressions are evaluated with the help of a stack, and its corresponding stack semantics [162], we see that even here apparently recursive uses of procedure names may disappear. In the stack semantics identifiers denote stack positions rather than values, and references to the value may be constructed before the value is worked out. According to Stoy “this allow a recursive use of a procedure name, for example, to be treated just like any other free variable variable of the procedure body” [162]. We have here a telling illustration of the interchangeability of dynamic lookup operations and fixed-point constructions: thus, “recursive procedures... no longer involve \texttt{fix} explicitly in their semantics.” [162]. In our view this fact extends to the semantic of objects defined as records or closures, as will be explained later.

The interpretation scheme presented above admits forward references in the body of functions, which promotes modularity and supports incremental, interactive programming. The body of a function may refer to the function itself, or contains
names that are yet undefined. Procedures may be redefined without affecting old references to it. These features are basically dynamic and it is hard to give them a meaning in a fixed-point semantics.

One point worth attention in this context concerns the relation between function calls and synchronisation. The evaluation of expressions in a functional language is done basically by searching for the value associated with identifiers in a specified environment $\rho$, and in case the expression being evaluated is a function application of type $f(y)$, substituting the function identifier $f$ by its value and then evaluating the latter in $\rho$ extended with the binding of the formal parameters $\bar{x}$ to the result of the evaluation of $y$ in $\rho$. This closely resembles the way functions are modelled in process algebra formalism. Usually, a synchronous message $f(y)$ is offered for synchronisation along $f$ with some agent of type $f(\bar{x}).P$ representing the function. After synchronisation, the agent $P[\bar{x}/y]$ executes, which corresponds to the evaluation of the function call. The difference is that in the functional setting it is assumed that there is one and only one definition within the scope of any identifier, whereas in process calculi there may be several different agents representing a certain function. Also, environments or frames represent contexts delimiting the meaning, i.e. the binding capabilities, of identifiers, whereas no notion of locality is present in for instance the $\pi$-calculus, although it is becoming increasingly common in other formalism and in extensions to the $\pi$-calculus.

A second important point concerns the relation between the kind of environments discussed above, which are based on what is known as "closures" of function, and the way objects themselves have been modelled as closures. This approach resulted in the introduction of a fixed-point semantics to give meaning to features in objects that do not seem to have much to do with recursiveness, and illustrates the issue of recursiveness versus self-references discussed above.

As pointed out in [59], denotational semantics usually abstracts away from the running process, and properties that are associated with running processes in a natural way loses this naturalness in a denotational translation. It also seems to introduce features which are not present in the language itself, as recursion. A good illustration of this point is offered by the account given to recursion by Dami in [40]. Dami proposes a classification of "various strategies for encapsulation and inheritance, based on their different uses of recursion", where "the crucial point to observe is where fixed-point operations occur." He distinguishes four possibilities. The first one is to apply a single recursive abstraction involving both instance variables and methods, which makes it hard to capture inheritance. The second possibility is to separate the class of objects from the instance variables. In this approach, methods in a class may access each other because the class is recursively bound to itself, and objects are endowed with a separate level of recursion for accessing instance variables; the result is static binding of methods. A third possibility is to remove recursion at the class level; in this case classes are referenced by objects through a class field, and methods need not be bound to each other. A fourth possibility is to apply a fixed-point operation at method invocation by passing a self parameter to the method. Other possibilities may exist, although Dami conjectures that the four cover all existing forms of inheritance.

This proliferation of fixed-point constructs is bewildering. Like most authors, Dami considers recursion an "essential ingredient of object-oriented programming." The reason he gives is that "methods call other methods, classes refer to other classes, object modify their state and return an updated object: all this is done using recursion." Nevertheless, as the author admits, "the exact role of recursion is difficult to understand, and many variations appear, both in programming languages and in the model that try to explain them." But this puts into question the supposed
essentiality of recursion concerning objects. If we view objects as processes, we
could say instead that “all this is done by referencing”, not recursion. This would
eliminate with the “many variations” referred to by Dami, and results in a simpler
and more uniform model for objects than the functional one.

Functions Without Values: the Continuation-Passing Style

Steele and Sussman [161] show that values may not be essential even in functional
languages. Recursion may be reduced to tail-recursion, or “tail-transfer”, a term
that describes this feature better. Tail-recursion “can be used for transfer of control
without making any commitment about whether the expression expected to return
a value.” The authors argue that the continuation-passing style makes explicit the
necessary steps to compute the value of an expression, in contrast to languages
such as FORTRAN. This brings functions a step closer to processes, a fact that was
already apparent to Milner [110], according to whom the technique of continuations
is due independently to Morris and to C. Wadsworth and reported by Reynolds [143].
The translation of the value-free \( \lambda \)-calculus to the \( \pi \)-calculus presented by Milner in
[110] can be said to follow this continuation-passing style. Nevertheless, in [44] the
origin of the concept is traced back to Van Wijngaarden, who during a discussion
at the IFIP Working Conference on Formal Language Description Languages in
1964 explained a “trick” by which one may “find that the actual execution of the
program is equivalent to a set of statements; no procedure ever returns because it
always calls for another one before it ends, and all of the ends of all the procedures
will be at the end of a program... If one procedure gets to the end, that’s the end
of all... That means you can make the procedure implementation so that it does
not bother to enable the procedure return.”

In the presence of intermediate results, the opposite capability, i.e. the capability
to hide intermediate results rather than pass it explicitly in continuations, is often
viewed as an improvement in the science of programming. According to Andrew
Appel [18] “the beauty of FORTRAN - and the reason it was an improvement over
assembly language - is that it relieves the programmer of the obligation to make up
names for intermediate results.” Nevertheless, in [161] it is shown how this may be
done in a functional language without the need to “make up names”. The import-
ance of continuations was already apparent to Church, according to [161], where
we may also read that this is “the only way to get anything at all accomplished
in pure \( \lambda \) calculus”. Nevertheless, as pointed out by Boudol [26] “a large part of
the development of the \( \pi \)-calculus as a programming language consists in intro-
ducing derived forms to get around the inconveniences of a “continuation passing
style”. Maybe this fact reflects the inadequacy of the \( \pi \)-calculus as a programming
language, and its adequacy as a semantic foundation for distributed and concur-
rent computation, where intermediate results cannot be abstracted away. We could
even turn things around and say that a large part of the development of a theory
for concurrent process consists in the introduction of primitive forms that make
intermediate results explicit, thus emulating the “continuation passing style” and
making explicit the necessary steps of computation. Cf. Milner’s claim that “the
memory-access history of programs must be included in their meanings if we are to
be able to compound the meaning of a parallel program from the meanings of its
components”, since the memory-access history must include all intermediate results
of computations.

It is thus possible to do without names and still uphold a continuation passing
style, at least for sequential languages, as exemplified by functional languages in
continuation-passing style. Nevertheless, as pointed out in [160], one of the basic
primitive operations common to all high-level programming languages are environ-
ment operations, and the operation of assignment may be mimicked in a functional program as an environment operator.

Environment operators may be regarded as operators that alter "the contents of a named location", or cause "the value associated with a name to be changed." [160] Thus, names are still there, but are not immediately visible, acting, as it were, behind the scene. Nevertheless, it is declared that "it is not names which are important to the computation, but rather the quantities." Since the scoping discipline is static, names may be alpha-converted when shadowed by new declarations of the same name, thus guaranteeing the "uniquisation" of variable names, exactly as if created by a new-operator similar to the one defined in the \(\pi\)-calculus. In this way, names may be considered "merely a convenient textual device for indicating the various places in a program where a computed quantity is referred to" [161]. It would thus be possible to "draw arrows instead", as in a data flow diagram". Names are in any case "eliminated at compile time". However, this is only true in the absence of dynamic binding. Since in the presence of static scoping the binding must occur at a fixed place relative to the environment, a compiler could determine the appropriate offset and no search for a variable's bindings would be necessary. Consequently, names would be also unnecessary.

The reason for the fact that in the early days of functional programming most systems adopted LISP-like dynamic binding, and current systems largely prefer static scoping, is quite technical in nature, and usually not related to semantic issues. Dynamic binding appeared first as a bug, as we pointed above, and its solution led to the implementation of closures. In [161], it is claimed that the primary reason for employing it might have been the introduction of stack hardware at the time LISP was being developed, since the structure of the stacks in dynamic binding is similar to the control stack in structure. However, there were two difficulties, according to [161]: the FUNARG problem [118], which amounts to the fact that lexical scoping is desired for functional arguments, and the binding stack problem. The latter refers to the fact that the binding stack may grow arbitrarily deep since dynamic variables must be pushed onto the bindings stack before function call, and a return address must be provided for popping those variables. This contradicts the view proposed in [118] about function calls, and lexical scoping was thus preferred.

Dynamic binding and static binding are not completely antithetical. In fact, there is a well-known translation from programs using dynamic binding to programs using static binding, the dynamic-environment passing translation [140]. The translation is achieved by adding a function's dynamic environment as an extra argument to the function. Thus, references to dynamic variables may be translated into lookup operations in the current dynamic environment. What seems essential here is not the kind of scoping discipline adopted, but the fact that all computation requires some kind of environment and lookup operations.

### 6.2.3 Denotational Semantics of Objects as Closures

A closure may be defined as a data structure containing functions with some local bindings, and may be identified with the notion of environment. This notion was originally devised for encapsulating side effects in functional programming, but it was soon found useful for describing the semantics of object-oriented languages.

In [142, 84] a denotational semantics in terms of closures for a series of small abstract objects-oriented language is presented. For a comparison of this semantics with others see [27]. Since our purpose here is to concentrate upon the use of self and the fixed point operator for objects, we will discuss here only one of these languages, called Classtalk, a language in which classes can be defined and objects
created as instances of classes, but where there is no inheritance. The semantics interprets objects as message environments, i.e., environments binding message names to methods.

The language contains two kinds of syntactic objects, variables and expression, and a few expression constructs. These are the following:

\[
\begin{align*}
x, y & \in \text{variable} \\
e & \in \text{expression} \\
e ::= x \\
e ::= \text{valof } e \\
e ::= x := e \\
e ::= \text{let } x = e_1 \text{ in } e_2 \\
e ::= \text{local } x; e \text{ end} \\
e ::= \text{class } (x_1, \ldots, x_n) \{ m_1(y_1) = e_1, \ldots, m_k(y_k) = e_k \} \\
e ::= \text{new } e_c \\
e ::= e_o.m(e_a) \\
e ::= \text{hierarchy } x_1 = e_1, \ldots, x_n = e_n \text{ in } e
\end{align*}
\]

Here \text{valof} denotes a dereferencing operator to access the contents of a location. Mutable variables in local contexts are expressed by the \text{local} construct, class by \text{class}, and instance of objects by \text{new}. The local variables of objects are \(x_1, \ldots, x_n\) in the definition of \text{class}. The syntax for message sending is \(e_o.m(e_a)\), where \(e_o\) is the receiving object, \(m\) the message, and \(e_a\) the argument expressions.

The top level of the program is expressed by a construct of type

\[
\text{hierarchy } x_1 = e_1, \ldots, x_n = e_n \text{ in } e,
\]

where expressions \(e_i\) are restricted to be class-expressions, and the free variables to be among \(x_1, \ldots, x_n\).

The semantic domains are the following:

\[
\begin{align*}
v, w & \in \text{val} = \text{basicval} + \text{loc} + \\
\eta & \in \text{env} = \text{variable} \to \text{val} \\
\sigma & \in \text{state} = \text{loc} \to \text{val} \\
o & \in \text{objectval} = \text{menv} \\
p & \in \text{menv} = \text{message} \to \text{method} \\
\mu & \in \text{method} = \text{state} \to \text{val} \to (\text{val} \times \text{state}) \\
\xi & \in \text{classval} = \text{state} \to (\text{menv} \times \text{state})
\end{align*}
\]

Environments and states are finite functions. The semantics interprets objects as message environments, i.e., environments binding message names to methods. Objects are created by invoking the class construct.

The meaning of an expression, \([e]\) is of type

\[
\text{env} \to \text{state} \to \text{val} \times \text{state}
\]
Thus, $[[e]]\eta\sigma$ yields a value of type $(v,\sigma')$. The bindings of free variables in $e$ are defined in $\eta$. Variables may be bound to either values or locations, and the contents of locations are obtained from $\sigma$.

The semantic definitions are as follows:

$$[[x]]\eta\sigma \quad = \quad \langle \eta x, \sigma \rangle$$
$$[[\text{valof } e]]\eta\sigma \quad = \quad \text{let } \langle \alpha, \sigma' \rangle = [[e]]\eta\sigma \quad \text{in } \alpha \in \text{loc} \rightarrow \langle \sigma'\alpha, \sigma' \rangle; ?$$
$$[[x := e]]\eta\sigma \quad = \quad \text{let } \alpha = \eta x, \langle v, \sigma_1 \rangle = [[e]]\eta\sigma \quad \text{in } \alpha \in \text{loc} \rightarrow \langle v, \sigma_1[\alpha \rightarrow v] \rangle; ?$$
$$[[\text{let } x = e_1 \text{ in } e_2]]\eta\sigma \quad = \quad \text{let } \langle v_1, \sigma_1 \rangle = [[e_1]]\eta\sigma \quad \text{in } [[e_2]](\eta[x \rightarrow v_1]); \sigma$$
$$[[\text{local } x; e \text{ end}]]\eta\sigma \quad = \quad \text{let } (\eta_0, \sigma_1) = \langle \eta_1, \sigma_1 \rangle = \langle \alpha = \text{newloc } \sigma, \sigma_1 = \text{extend } \sigma \alpha, \eta_0 = \eta_{\perp}[x \rightarrow \alpha] \rangle \quad \text{in } [[e]](\eta; \eta_0)\sigma_1$$

The symbol $?$ denotes an error value, and expressions of type $E \rightarrow E_1; E_2$ mean if $E_1$ then $E_2$ else $E_2$. The notation $\eta[x \rightarrow v]$ or $\sigma[x \rightarrow v]$ means a copy of $\eta$ resp. $\sigma$ that maps $x$ to $v$ and leaves everything else unchanged. The notation $\eta; \eta'$ means updating of $\eta$ with all bindings of $\eta'$. Similarly for $\sigma$. The symbols $\eta\perp$ and $\sigma\perp$ denotes the empty environment resp. the empty state.

The semantics of local involves the expression `alloc $\sigma$`, which is defined as follows:

$$\text{alloc } \sigma x \quad = \quad \text{let } \alpha = \text{newloc } \sigma \quad \text{in } \langle \eta_0, \sigma_1 \rangle$$

The meanings of class, new, message send and hierarchy are defined as follows:

$$[[\text{class } \{ \text{m_1(\overline{m}) = e_1 \} }]]\eta\sigma \quad = \quad \langle \sigma' \rangle \text{let } (\eta_0, \sigma_1) = \text{alloc } \sigma' \overline{m} \quad \rho = \text{fix } (\lambda \rho. \rho_{\perp}[m_i \rightarrow (\lambda \sigma. \lambda \overline{m}. [[e]](\eta; \eta_0[m_i \rightarrow \overline{m}, \text{self } \rightarrow \rho])\sigma)); \sigma)$$

$$[[\text{e.e.m(\overline{m})}]]\eta\sigma \quad = \quad \text{let } \langle \rho, \sigma_1 \rangle = [[e]]\eta\sigma \quad \langle \overline{\sigma}, \sigma_2 \rangle = [[\overline{m}]]\eta\sigma_1 \quad \text{in } \rho m\sigma_2\overline{m}$$

$$[[\text{new } e_1]]\eta\sigma \quad = \quad \text{let } \langle \xi, \sigma_1 \rangle = [[e_1]]\eta\sigma \quad \text{in } \xi\sigma_1$$

$$[[\text{hierarchy } x_1 = e_1, \ldots, x_n = e_n \text{ in } e]] = \quad \text{let } \phi = \lambda \eta. \eta_{\perp}[x_i \rightarrow \text{fst}([[e]]\eta\sigma_{\perp})] \quad \eta = \text{fix } \phi \quad \text{in } [[e]]\eta\sigma_{\perp}$$

As we may see from the definitions, the denotation of objects consists of an environment binding message names to methods. Thus, the object’s local environment
is excluded from its denotation. The message environment is a closure since it absorbs the local environment $\eta_0$, which is an exclusive window to the locations in $\sigma$ representing the local environment of the object.

The special variable `self` is used for recursive references to an object’s messages in the methods defining those messages. In the definition of the meaning for class expressions, \( \text{self} \rightarrow \rho \) indicates that the variable `self` is bound to the message environment $\rho$, making the definition of the latter recursive. The environment is thus defined by a fixed point and the variable `self` is bound to that environment at the time of object creation. The behaviour of the object is thus fixed at this point, precluding dynamic inheritance, as in `SMALTTALK-80` or the virtual functions of C++. In dynamic inheritance the variable `self` cannot be bound directly by the class expression, as is the case here, but only at the time of object creation. Fixed-point constructions usually lack dynamicity since it “freezes” the resulting construct. Furthermore, instance variables defined in a class cannot be accessed by its subclasses. The definition of `class` must thus be redefined to make place for inheritance.

Another fixed point is involved in the definition of hierarchy, the global environment; there may be mutual recursion in the definition of classes. This environmental fixed-point also freezes the environment being defined by the fixed point operation, precluding its dynamic extension with new classes, or subclasses if inheritance is introduced.

The reason for defining the meaning of an object recursively is not immediately apparent. Recursive invocations to methods defined in the same class could be regarded as local. This is commonly the way closures are defined. In functional programming, a call to a function occurring in a closure is done typically by looking up the value of the function, a lambda-expression, in the environment defined by the closure. The closure itself is not invoked, it is a passive data structure supporting computation, usually a hierarchy of frames. As we saw above, this data structure is cyclic only for reasons of efficiency.

An environment may be also regarded as a value, and can be passed around, but usually this is done by referencing. In the definition above, $\rho$ is not a reference, but a variable that takes environments as values.

A good illustration of how fixed point constructions vanish in the presence of dynamic lookup of identifiers is Kamin’s semantic description [83], upon which the semantics we are discussing here was based. In contrast to Reddy’s semantics, Kamin interprets objects denotationally as pairs of local environments and references to class denotations. The denotations of classes are defined independently of the objects. Local environments are thus part of the denotation of objects, but what interests us here is the fact that instead of defining objects as environments binding message names to methods, what we get instead are objects referencing classes in their denotation.

The semantic domains are the following, in the version given in [142], where they appear subscripted by $K$ as an indication that they refer to Kamin’s semantics:
The denotation of objects contain now a local environment representing the state of the object, and a classname, an identifier denoting the class to which the object belongs. From the point of view of the user an object is now a member of a class determining the set of messages it may respond to, and a local state determining its specific behaviour, i.e. its behaviour in relation to other objects of the same class.

The denotation of a class in Kamin’s semantics contains the local instance variables in addition to the message environment, and is no longer a function yielding an object when given a state. Message environments are now associated with classes, not objects. The meaning of a class expression in Kamin’s semantics is the following:

\[
\llbracket \text{class}(\bar{x}) \rrbracket \{ m_i(\bar{y}) = e_i \} \eta \sigma = \\
\begin{align*}
\text{let } & \rho = \rho_1[ m_i \to \lambda \sigma. \lambda o_r. \lambda \bar{m}. \text{let } \{ \eta_r, c_r \} = o_r \\
& \text{ in } e_i \} \{ \eta, \eta_r(\bar{m} \to \bar{m}, \text{self} \to o_r) \} \sigma \\
\end{align*}
\]

The denotation of objects is defined as instantiation of a class with the help of an auxiliary function lookup, which looks up the message environment of a class \( c \) in an environment \( \rho \):

\[
\text{lookup } c \ \eta = \ \text{let } (\bar{x}, \rho) = \eta \ c \ \text{in } \ \rho
\]

\[
\llbracket \text{new } c \rrbracket \eta \sigma = \text{let } (\bar{x}, \rho) = \eta c \\
\{ \eta_o, \sigma_1 \} \text{ alloc } \sigma \ \bar{x} \\
\text{in } (\eta_o, c), \sigma_1
\]

The denotation of methods differs too: It now takes an implicit argument denoting an object, the receiver object, since the denotation of a method is defined in the context of a class. This was not necessary in the previous semantics since a message environment there is associated with the object which is defined as a fixed point operator involving the variable self. This amounts to an orthogonalisation of the relation between objects and methods. Shortly, we may regard methods as functions that are polymorphic on the implicit argument denoting the receiver of the message, instead of regarding objects as polymorphic on the class it belongs, according to the definition of inclusion polymorphism given in [30]. In Kamin’s semantics references to self are interpreted as references to the implicit argument. We have here another example illustrating the equivalence of self-referential constructs and dynamic binding and lookup of values. As pointed out in [142], “there is a kind of dynamic binding involved in the interpretation of method expressions. The environment \( \eta_r \) is obtained dynamically as part of the parameter \( o_r \) to the method, but is used for binding the free variables in \( e_i \).”

The semantics of message and the hierarchy construct are as follows:
\[
\llbracket e \cdot m(\rho) \rrbracket \eta \sigma = \text{let } \langle \langle \eta, c_o \rangle, \sigma_1 \rangle = \llbracket e \rlbracket \eta \sigma \\
\rho = \text{lookup } e, \eta \\
\langle \eta, \sigma_2 \rangle = \llbracket e \rlbracket \eta \sigma_1 \\
in \rho \sigma_2 \langle \eta, c_o \rangle \eta \sigma
\]

\[
\llbracket \text{hierarchy } c_1 = e_1, \ldots, c_k = e_k \text{ in } e \rrbracket = \\
\text{let } \phi = \lambda \eta \eta_\perp [c_i \rightarrow btl(\llbracket e_i \rrbracket \eta \sigma_\perp)] \\
in \llbracket e \rrbracket \eta \sigma_\perp
\]

Thus, message transmission implies a kind of polymorphic method application. First the local environment and class reference of the object receiving the message is evaluated; then the message environment of the corresponding class is looked up; afterwards, the arguments are evaluated; and finally, the method corresponding to the method environment evaluated in the lookup operation is applied.

As we may observe, no recursion is involved at this stage. The reasons, as pointed out in [142], is (i) the fact that methods are defined as functions of the receiver objects, which amounts to a “delayed self-reference”, a dynamic feature; (ii) the use of classnames which “also involves a delayed self-reference”. Direct use of the environment \( \rho \) instead of classname \( c \) would result in the introduction of “implicit recursion”, as pointed out in [142], yielding the expression \( \rho \sigma_2 (\eta, \rho) \eta \), which involves a self-application of \( \rho \). We see here how the interplay of names and dynamic delayed references through lookup operation eliminates explicit recursion which once again puts into question the appropriateness of the term recursion in this context. Denotational semantics has a bias towards functional notions such as values, and against syntactic terms such as names. We claim that recursion and fixed-point semantics are not always adequate concepts to describe the meaning of programming language constructs.

In functional languages the meaning of recursively defined functions may adequately be described by a fixed point operator. The question is whether the same is true of objects. The result of fixed point operations on objects yields very little information, in contrast recursive defined functions. Usually it consists only of an early expansion of the terms occurring in the methods.

As Reddy points out, Kamin’s semantics involves “unconventional use of the syntactic domain classname” in the semantic domain of objectal, arguing that traditionally “denotational descriptions only use names as input domains of environments.” However, the phenomenon appears, not surprisingly, in connection with the use of reflexive domains for the definition of environments in languages with dynamic scope rules. Thus, in [162], environments \( U \) are defined as domains \([\text{Id} \rightarrow D]\), where \( D = [U \rightarrow B] \), \( B \) denoting a domain of basic values. The values denoted by identifiers in an environment are of functionality \( U \rightarrow B \) rather than simply \( B \). Thus, the final value of an identifier is “delayed” or obtained dynamically from the current environment during execution time. Another telling example involving an extensive use of “unconventional” domains appears in the denotational semantics of the language POOL [17], which we discuss in Section 6.2.4.

The “delayed recursion”, according to Reddy [142], is captured by the environmental fixed point in the semantics of the hierarchy construct. Nevertheless, the function of this fixed point operator is rather to “construct” the global environment by resolving mutual references made in the body of the distinct defined classes. This fixed point yields a static structure resembling a compiled program, and is thus adequate only under the “closed world assumption.” It “freezes” the global environment, which
may no longer be extended with new classes or even subclasses of the classes already
defined unless the environment is “recompiled”. Furthermore, it does not affect the
meaning of objects, as in Reddy’s semantics, which are supposed to be recursive in
nature. The meaning of objects is \( (\eta_o, c) \), where \( \eta_o \) is a local environment created
by alloc that does not depend on the the global environment, and \( c \) is only a name.
The meaning of the latter is only resolved dynamically by a lookup operation that
does not discriminate between self references and references to any other objects or
classes.

Since the message environment of the object is obtained dynamically from the
implicit argument denoting the receiver, the semantics of inheritance in Kamin’s
semantics is extremely simple, in contrast to Reddy’s semantics (for a definition of
the latter we refer to \([142]\)):

\[
[[\text{subclass } c(v) \{ m(v) = e \} ]] =
\text{let } \langle \eta, \rho, c \rangle = \eta_o
\rho = \rho_e \downarrow m \to \lambda \eta_o \cdot \lambda \eta_r \cdot \lambda \eta_r \cdot \text{let } \langle \eta_r, c_r \rangle = \eta_r
\text{in } \langle e[[\eta_o, \eta_r, \eta_r \rightarrow \rho, \text{self } \rightarrow \rho_r] \rangle
\]

6.2.4 Denotational Semantics of POOL

An important study that shows a denotational semantics for a parallel object-
oriented language was presented by America et al. in \([17]\). In spite of its merits,
this approach illustrates in a parallel setting some of the difficulties we have already
pointed out in relation to this kind of semantics when it is applied to give meaning
to object-oriented concepts (Section 6.2.3).

The language POOL describes systems that consist of a collection of communica-
ting \textit{objects} with internal activities running in parallel. Objects are dynamic
toies containing \textit{data} stored in \textit{variables} as well as \textit{methods}. Objects can be
created dynamically, and each one contains a \textit{body} representing internal activity.
The body of objects consists of statements to be executed \textit{sequentially}, parallelism
being restricted to the concurrent execution of the bodies of the different objects.
Concurrency is obtained via \textit{message passing}. Message acceptance gives rise to a
\textit{rendezvous} between sender and receiver. POOL programs are called \textit{units}, and
consist of a finite set of class declarations of type

\[
U ::= \langle C_1 \leftarrow d_1, \ldots, C_n \leftarrow d_n \rangle \ (n \geq 1)
\]

A unit is executed by creating an instance of the last class defined in the unit.
\( C_1, \ldots, C_n \) are elements of a set of class names, and \( d_1, \ldots, d_n \) are class definitions

\[
d ::= \langle (m_1 \leftarrow \mu_1, \ldots, m_n \leftarrow \mu_n) \ (n \geq 1) \rangle
\]

where \( m_1, \ldots, m_n \) are elements of a set of method names and \( \mu_1, \ldots, \mu_n \) are method
definitions

\[
\mu ::= \langle (u_1, \ldots, u_n), e \rangle.
\]

Here \( e \) is an element of a set of expressions \( \textit{Exp} \) that may include variables, message
sending, method invocation, object creation, equality of expressions, the special
variable \textit{self}, a statement followed by another expression, or a standard object (in-
tegers, booleans, nil). Statements include assignments, answers to message sending,
expressions, sequences, and \textbf{if}, \textbf{do} and \textbf{sel} (select) statements. For details we refer
to \([17]\).
The semantics of the language is based on a mathematical domain of *processes* which is obtained as a solution to a reflexive domain equation over a category of complete metric spaces. It assumes the following syntactic domain of names:

- **IVar** instance variables
- **TVar** temporary variables
- **CName** class names
- **MName** object names

where **TVar** is used for keeping track of temporary variables in method calls. A function

\[ \tau : A\text{Obj} \to C\text{Name} \]

is also assumed, assigning to each object \( \alpha \in A\text{Obj} \) the class to which it belongs, or rather the name of the class.

Furthermore, the semantics assumes also a function

\[ \nu : \mathcal{P}_{\text{fin}}(A\text{Obj}) \times C\text{Name} \to A\text{Obj} \]

where \( \mathcal{P}_{\text{fin}}(Y) \) is the set of finite subsets of \( Y \), \( \nu(X, C) \notin X \), \( \tau(\nu(X, C)) = C \). The function \( \nu \) simply yields a fresh object name, i.e. an object name that does not occur in its first argument.

The set \( \text{Obj} \) of objects is defined by

\[ \text{Obj} = A\text{Obj} \cup \mathbb{Z} \cup \{tt, ff\} \cup \{nil\} \]

The set of states \( \Sigma \), is defined by

\[ \Sigma = (A\text{Obj} \to \text{Ivar} \to \text{Obj}) \]
\[ \times (A\text{Obj} \to \text{TVar} \to \text{Obj}) \]
\[ \times \mathcal{P}_{\text{fin}}(A\text{Obj}) \]

The states and its components are denoted by \( \sigma = <\sigma_1, \sigma_2, \sigma_3> \). The first component, \( \sigma_1 \), stores the values of the instance variables for each object in the system; the second component, \( \sigma_2 \), stores the values of the temporary variables for each object; the third component, \( \sigma_3 \), denotes the object names currently in use. As we may observe here, objects are represented as elements of functions in a determined state. No other meaning is assigned to objects in this semantics.

The meaning of *processes* \( P \) is the complete ultra-metric space satisfying the following equivalences:

\[ P \cong R_0 \cup d_{1/2}(\Sigma \to \mathcal{P}_c(\text{Step}_P)) \]
\[ \text{Step}_P = (\Sigma \times P) \cup \text{Send}_P \cup \text{Answer}_P \]
\[ \text{Step}_P = \text{Obj} \times \text{MName} \times \text{Obj}^* \times (\text{Obj} \to P) \times P \]
\[ \text{Answer}_P = \text{Obj} \times \text{MName} \times (\text{Obj}^* \to (\text{Obj} \to P) \to P) \]

The subexpression \( d_{1/2} \) above needs not concern us here. \( \text{Obj}^* \) denotes the set of finite sequences of object names. A process is either \( p_0 \), the terminated process, or a function from \( \Sigma \) to \( \mathcal{P}_c(\text{Step}_P) \), where \( \mathcal{P}_c = \{X : X \subseteq M \land \text{closed}\} \) according to a
metric whose definition we may safely omit here. The intuitive meaning is that the process \( p \) in state \( \sigma \) has a choice among the steps of \( p(\sigma) \). A step consists either of some action leading to a new state and a resumption of the action (a new process), or an attempt at communication, either sending or answering a message. Message sending requires the name of the targeted object, a method name, the arguments (object names), a dependent resumption indicating the steps that the sender will take after receiving the reply to the message, and finally an independent resumption denoting the action to be taken immediately after message sending and independently of any reply. A message answer requires the name of the object willing to accept a message, the name of the method to be invoked, and a resumption to the answer step. The latter consists of the arguments to the method and the dependent resumption of the sender, resulting in a process that denotes the resumption of the sender and the receiver together, composed in parallel with the independent resumption of the sender. Parallel composition \( p \parallel q \) is modeled by interleaving, and consists of all possible steps of \( p \), all possible steps of \( q \), and the communication steps between \( p \) and \( q \).

Environments are defined by

\[
\text{Env} = (A \text{Obj} \to P) \times (\text{MName} \to A \text{Obj} \to \text{Obj}^\eta \to (\text{Obj} \to P) \to P)
\]

The environment contains all information about the definitions present in a unit \( U \). The first component \( A \text{Obj} \to P \) is a function that, given an object name, yields a process representing the execution of the body of the object, which depends on the class of the object that can be determined by the function \( \tau \). Note that \( \tau \) yields a class name, a reference, not the body itself. The latter will be determined by the semantics of a Unit, explained below. The second component takes a method name and an object, yielding the execution of the method by the object, which requires the list of arguments passed to the method and an expression continuation, and yields a new process expressing the meaning of the execution of the method.

The semantics of expressions is given by a function \( [[\ldots]]_S \) of type

\[
[[\ldots]]_s : \text{Stat} \to \text{Env} \to A \text{Obj} \to \text{Cont}_S \rightarrow P
\]

where \( \text{Stat} \) is the set of statements, \( \text{Env} \) the set of expressions, and \( \text{Cont}_S = P \) the set of continuation of statements. Given a statement, an environment, the object executing the statement, and a process representing the continuation, a process will result. For expressions we have

\[
[[\ldots]]_E : \text{Exp} \to \text{Env} \to A \text{Obj} \to \text{Cont}_E \rightarrow P
\]

which is similar to a statement except for the continuation, which in this case requires as input the value of the expression being evaluated, i.e \( \text{Cont}_E = \text{Obj} \to P \). We leave out the definition of \( [[\ldots]]_S \) and \( [[\ldots]]_S \).

The semantics of a Unit \( U \), where

\[
U ::= \langle C_1 \leftarrow d_1, \ldots, C_n \leftarrow d_n \rangle \ (n \geq 1),
\]

depends on the environment that contains information about the class and method definitions of \( U \). The meaning of \( U \) is given by a fixed point construction, a function \( \Phi_U : \text{Env} \to \text{Env} \). If an environment \( \gamma \) is denoted by \( \langle \gamma_1, \gamma_2 \rangle \), and \( \Phi_U(\gamma) \) by \( \gamma \), then

\[
\gamma_1(\alpha) = [[\alpha]]_S(\gamma)(\alpha)(p_0)
\]

where
Thus, \( \tilde{\gamma}_1 \) provides a process representing the body of the object. The second component is defined by

\[
\tilde{\gamma}_2 (m)(\alpha)(\overline{\beta})(f) = \lambda s. \{ e \}_{E}(\gamma)(\alpha)(\lambda \beta. \lambda \sigma. \{ \sigma', f(\beta) \})
\]

where \( m \in MName, \alpha \in AObj, \overline{\beta} \in obj^*, f \in Obj \to P, C = \tau \alpha, \) and

\[
\begin{align*}
U &= \langle \ldots, C \leftarrow d, \ldots \rangle \\
d &= \langle \ldots, (\ldots, m \leftarrow \mu, \ldots) \ldots \rangle \\
\mu &= \langle u_1, \ldots, u_n, e \rangle \\
\sigma' &= \langle \sigma_1, \sigma_2 \{ h/\alpha \}, \sigma_3 \rangle \\
\overline{\beta} &= \langle \beta_1, \ldots, \beta_n \rangle \\
h(u_i) &= \beta_i \text{ for } i = 1, \ldots, n \\
h(u) &= \text{nil } \text{ for } u \not\in \{ u_1, \ldots, u_n \} \\
\sigma &= \langle \sigma_1, \sigma_2 \{ \sigma_2(\alpha)/\alpha \}, \sigma_3 \rangle.
\end{align*}
\]

When \( \tilde{\gamma}_2 \) is applied to a method \( m \) and object \( \alpha \), the result is a state transformation initialising the variables of \( \alpha \) and the semantic value of the expression \( e \) used in the definition \( \mu \) of \( m \). After execution, the current environment before the execution of \( e \) is restored. Finally, the meaning of \( U, \gamma_U \), is defined as

\[
\gamma_U = \text{Fixed Point}(\Psi_U)
\]

whose existence is proved in [142].

An observation is in place here concerning the “unconventional use” [142] of several syntactic domains (see Section 6.2.3), i.e., used for other purposes than as “input domains of environments.” What we see here is an extensive use of the syntactic category of names as references to other entities such as classes or objects, thus highlighting their intrinsically dynamic capabilities. It seems that the referential function of names becomes increasingly indispensable with increasing concurrency and distribution. The fact that names seem to be unavoidable even in the context of denotational semantics gives further proof of the adequacy of the object paradigm for describing communication and concurrency. In fact, when applied to concurrent systems denotational descriptions tend to introduce features which feel natural when they appear in the context of the object paradigm, but rather awkward when they are seen in the context of a denotational semantics.

The most conspicuous feature of this semantics is thus the “unconventional” use of syntactic categories and referencing, as mentioned above. Another fact worth attention here is that objects are not defined as fixed points, and in fact not defined at all. The fixed point operation we see in this semantics is applied at a construct which plays the same role as the one called hierarchy in Kamin’s semantics (see Section 6.2.3), and concerns recursive invocations of one class by another, rather than recursive invocations of an object by itself. The pseudovariable \texttt{self} is defined as a proxy for the name of the object, not the object itself. Its semantics is given by

\[
\llbracket \texttt{self} \rrbracket_{E}(\gamma)(\alpha)(f) = f(\alpha),
\]

158
i.e. the name of the object in question, \( \alpha \), is passed immediately as the argument of the continuation function \( f \), whereas the reference of object names are looked up in the environment (state).

Fixed point definitions yield constructs that may be characterised as closed systems. As conceded by the authors, the systems obtained by the semantics “are not expected to communicate with any environment,” and any attempts to do it are regarded as “unsuccessful attempts at communication.” Also, the concrete mathematical domain used, the space \( P \) of processes, “appears to be too complex”, but “if we want simpler (smaller) domains, we shall have to use different ones for different syntactic categories.” The authors believe this is is desirable, among other reasons because the semantics “does not provide a clear view of the basic concept of the language, the concept of an object... It would be nice to have a semantics in which the objects appear as building blocks of the system and in which their fundamental properties, e.g., with respect to protection, are already clear from the domain used for their semantics.”

Another shortcoming the authors point out is that the semantics provided is unlikely to be fully abstract. The reason is that “full abstractness assumes a notion of observable behaviour of a program and in the language as we have presented it, programs do not interact at all with the outside world.” These interactions are essential in open systems and require constructs that are at the base of nominal process calculi, whose semantics is defined exactly in terms of observable behaviour.

### 6.2.5 The \( \lambda \)-calculus with Names

The work of Laurent Dami about functional programming with dynamic binding [41] and the integration of functional and object-oriented approaches [40] is very relevant for us, since it explicitly gives an account of names as basic elements in communication systems. In his thesis [40] Dami claims that the functional approach toward object-oriented systems is inadequate because the standard \( \lambda \)-calculus lacks the extensibility property, i.e. the ability to extend a piece of software while remaining compatible with the original context in which it was defined. This extensibility property could be achieved by “the addition of names for controlling interaction, instead of using only positional information as in the standard calculus” (page 5 in [40]). This extension of the standard calculus enables the modelling of records, extensible data types and objects. Software composition is the object of Dami’s work, not concurrent programming, but he recognises that there are “some connections” between these subjects, claiming that some of the constructs he presents “have interesting prospects as far as concurrency modelling is concerned,” ([40] page 30).

The \( \lambda \)-calculus is usually extended with records [31] [34] in order to make room for extensibility and the related notion of subtyping. Records are similar to cartesian products, with the difference that projections are named. This implies that while cartesian product can be extended only unidimensionally, records and any orderings based on names can be extended in a partially-ordered fashion, an important feature which seems to contradict the claim that names are only notational convenience and that a name-free calculus, e.g. one that relies on positions rather than names, might offer the same functionality. This is apparently the thesis defended by Abramsky in [7], where processes are viewed as morphisms from and to process specifications, or types, that partition the interface of the process. A process is seen as concurrent module “with its interface to the environment represented as a bunch of wires.” Moreover, “by accessing this interface positionally, we can get combinators that let us do process algebra name-free.” Abramsky goes on to contrast “the name-free positional approach to the traditional use of names in process algebra as analogous
to positional access via n-tuples \(\langle a_1; \ldots; a_n \rangle\) vs. the labelled record representation

\[
[l_1 \Rightarrow a_1; \ldots; l_n \Rightarrow a_n].
\]

As we have seen, this squares badly with the conclusions reached by Dami. Dami argues that names are also useful in other contexts. Modules, for instance, are viewed as non-closed structures that “require the use of names to express how they should be globally linked together into a closed program” ([40] page 32), which accords with our characterisation of module linking as composition plus restriction. Another example are datatypes, which are usually built through constructors that are “identified by their name.” For our purposes, the most interesting example concerns the role played by the method names of objects or classes. According to Dami “names are naturally well-adapted to describe software objects: what is externally visible from an object is the set of names (messages) that it can handle.” This insight strengthens our argument concerning the adequacy of name-based calculi as a semantical basis for describing object-oriented concepts. To our knowledge this fact has not been recognised earlier. Dami himself seems to be aware of the novelty of these observations when he writes that they “led us to the feeling that there is something more fundamental about the use of names than what is captured by record systems, and motivated our quest for a new calculus” ([40] page 52). To our view what is fundamental about the use names is that they constitute the basis for communication in open concurrent systems, as well as for the object-based approach to computation (cf. Milner’s “object paradigm”).

Dami presents an extension of the standard \(\lambda\)-calculus called the \(\lambda N\) (\(\lambda\)-calculus with names), in which “names instead of positions for interaction between components introduces an extensibility dimension” ([40] page 63). This is done by allowing the names following the \(\lambda\) in an abstraction to be parameters, and application to be a binding of these parameters within a kind of partial closure. What we obtain is basically evaluation of expressions within an explicitly given environment. Variables are defined as names with a de Bruijn index, and a bind operation partly corresponds to application. A close operation is defined as a means for indicating the finalisation of a set of bindings. On top of the formalism, higher-level constructs are defined, such as extensible records, variants, extensible case statements, and dynamic associations of terms with environments. For details about the syntax of the \(\lambda N\) calculus, we refer to [40].

In contrast to the use of names in Lampings’s unified system of parametrisation (see Section 6.2.6) and Garrigue and Ait-Kaci’s label-selective \(\lambda\)-calculus (Section 6.2.6), both of which treat names and variables as orthogonal concepts, in the \(\lambda N\) calculus both concepts are unified through the use of de Bruijn indices. Furthermore, the label-selective \(\lambda\)-calculus does not support extensibility, and combines label-selection with currying.

On the theoretical foundation of the \(\lambda N\) calculus, a language called \(HOP\) was designed [41]. \(HOP\) is an extension of the functional model intended to offer new ways of composing software, to provide a framework for studying object-oriented programming features, and specially to integrate notions such as parametric polymorphism, higher-order composition, type inference and pattern matching with dynamic binding. \(HOP\) has no side effects and uses lazy evaluation.

According to Dami, “dynamic binding denotes a family of programming constructs where the runtime system includes some notions of ‘names’ and ‘environments’ (association from names to values), and where the operation of looking up some name in the environment is performed dynamically.” In fact, we may regard all lookup operations as dynamic in nature, and Dami himself defines it as “a runtime
operation which looks up some names in some environments.” In some cases the lookup operation may be “compiled away” by fixed references, which is possible only with static scoping. A crucial feature of dynamic binding is extensibility, which follows from the fact that an environment may be replaced by a larger environment without disrupting the lookup operations involving the original set of names, thus preserving the compatibility of the modified code with the original code. Such modifications are even possible in the absence of the source code because name lookup operations are performed by the runtime system, which is “a key factor” for the success of object-oriented programming.

An interesting observation made by Dami in this context [41] is that “programmers switching from the object-oriented world to the functional world often discovers unexpected frustrations, due to the absence of dynamic binding.” He gives as example the impossibility of writing a function like $\text{extract} X \leftarrow r.X$ in some versions of ML with records, where $\text{extract} X$ is a function that takes a record $r$ as argument and selects a field $X$ in $r$ “whatever the other fields of the record may be.” The reason is that “names of fields are not known at runtime, and cannot be dynamically looked up.” Therefore, the absence of dynamic binding restricts the expressiveness of a language. As a consequence, names cannot be “compiled away” or substituted by a reference to a fixed location or by position, without at the same time restricting the expressiveness of the language. Dami suggests a modification of the fundamental execution model of functional languages, and proposes $\lambda N$ as the new model. In this model, parameter passing is done on the basis of names that become part of the semantics of functions.

As noted in [42], computer systems should be able to interact even with “possibly unknown or weakly specified systems.” Open systems depend on dynamic binding, a concept which “seems to have been less investigated in theoretical studies”, according to Dami. Dami argues that the formal models developed so far for the open system paradigm incorporates dynamic binding features only implicitly and usually merged with other aspects of computation, and therefore are not adequate for the study of dynamic binding in an abstract setting. We believe that this is an accurate description of the current state of affairs, and consider it to be at least in part a result of a bias in favour of the functional paradigm.

Although Dami’s approach and conclusions are highly relevant to our subject, it still remains within the functional paradigm and is more concerned with flexible software construction than with open systems. Objects are not modelled in HOP, and Dami only claims that the approach that treats objects or classes as function records is amenable to the language.

6.2.6 Survey of relevant languages and formalisms

In this section we will give a short description of languages and formalism which we consider relevant in relation to the functional paradigm.

Pebble

In [90], a pioneering work on the use of modules in computation, Lampson and Burstall give an early model for features supporting the writing of programs in a modular fashion. The principal idea they wished to express was “the linking together of a number of modules into a larger program.” Program modules implement a collection of data types and procedures partially. In order to complete this implementation a module may require the implementations supplied by other modules according to its interface. These interfaces may consist possibly of data types, argument specification, result types, etc., but they are in any case invariably
denoted by names or free variables, which are basically the *means of communication* between modules. It may be compared to the channels along which processes may communicate in a process algebraic notation. Modules might have been developed at different times by different people, and “it is essential to be able to express easily different ways of connecting them together” [90]. This is basically the same kind of problem encountered by Milner, which he expressed as the “ability to define the operation of binding together two processes to yield another process representing the composite of two computing agents, with their mutual communications internalised.” [106]

Lampson and Burstall claim that linking must be treated in a systematic manner. To this end, they chose to use closures as the values of lambda expression, which amounts to using environments as first-class citizens. This is maybe the earliest appearance of first-class environments, although the authors did not call them so.

Bindings are considered in pebble as values, as well as types and declarations, which are also bindings. Closures are explicit values in the language, as well as the result of the evaluation of lambda expressions. Closures are the result of pairing bindings. The authors also introduce the notion of *parametrised modules*, functions from bindings to bindings.

For our purposes, the important feature is the basic role given to the names in bindings, which may be defined as a named collection of values. Like the names of the methods of an object, the names in a binding are not alpha-convertible, since they represent the means of communication of a module with other modules. For instance, a sorting module is represented in pebble as follows:

\[ \text{LET } \text{SortModule}(\text{lesseq} : (\text{int} \times \text{int} \rightarrow \text{bool}) \rightarrow (\text{issorted} : \ldots \times \text{sort} : \ldots) : \sim \ldots) \]

Here, a *SortModule* is a function from bindings where the free variable *lesseq* is bound to some value of type \((\text{int} \times \text{int} \rightarrow \text{bool})\), to bindings where *issorted* and *sort* are bound to values that appear after the symbol \(~\).

A module *Arith* may be defined approximately as

\[ \text{Arith} : \sim (\text{lesseq} : (\text{int} \times \text{int} \rightarrow \text{bool}) \sim \ldots, \text{add} : (\text{int} \times \text{int} \rightarrow \text{int}) \sim \ldots) \]

*Arith* is thus a module where *lesseq* is defined, as well as other variable names such as *add*, which are not needed by *SortModule*. The linking of both modules is represented by the expression

\[ \text{SortModule}(\text{Arith}) \]

In terms of process algebra, this construction would correspond to a simultaneous composition and restriction: a composition of two agents, *SortModule* and *Arith*, and restriction on the communication channels *lesseq* and *add*.

The function denoted by \(~\), the binding, also coerces the binding. Thus, in analogy with the type of objects as records, and subclassing or inheritance as record extension, the type of *Arith* is considered as a subtype of the type of the binding

\[ \text{lesseq} : (\text{int} \times \text{int} \rightarrow \text{bool}), \]

and thus the expression *SortModule*(Arith) is typable.

**Environments as First Class Objects**

In [55] a programming language is described, *Symmetric Lisp*, that treats environments as first-class objects. The conclusion of the authors is relevant for this
The authors claim that several apparently orthogonal tools are variations over the notion of environment, for instance abstract data types, records, closures, system interfaces. They show that “first-class environments bring about fundamental changes in a language’s structure”, and that “conventional distinctions between declarations and expressions, data structures and program structures, passive modules and active processes disappear.” This in our opinion is a reflection of the fact that a computation may be regarded basically as a continuous and dynamic transformation of a set of environments, and that the incorporation of environments explicitly in the language eliminates many dualities such as those represented by data and program structures, passive and active data structures, values and effects, etc. It is also significant that these distinctions also disappear when languages are translated into process algebra, for instance as in [109], chapter 8.

In Symmetric Lisp environments consist of name, private and alpha forms. Name forms bind names to the result of expressions, and are visible outside of the environment, in contrast to private forms, which also bind names to result of expressions but are visible only inside the environment. Alpha forms tie name and private bindings together into an environment. Names must be unique within a given alpha. In contrast to closures, the name fields of environments are accessible from the outside. In Symmetric Lisp “higher order, lexically scoped functions are supported in the presence of a dynamic binding discipline.” There is no difference between environments that serve as data structures and environments that are used in function application.

Environments may occur hierarchically inside other environments. If an expression refers to a name that is not bound within the innermost environment or alpha-form, a search is initiated in the first environment that encloses the innermost environment, and so on. This defines a dynamic binding discipline of the type we are going to exploit in the notation defined in Section 5.2. Environments may be also bound to names, and there are special forms with \textit{Q E} to allow an expression to \textit{E} be evaluated within environment \textit{Q}.

**Lamping’s Unified System of Parametrisation**

In [89] a formal system of parametrisation intended to express a series of apparently disconnected mechanisms is proposed by J. Lamping. Echoing the work on Symmetric Lisp [55] discussed in the previous section, Lamping exposes the inadequacy of the several mechanisms to express parametrisation, e.g. explicit access to environments, quotation, module facilities, and method lookup in object systems.

A system is parametrised, according to Lamping’s definition, “when it has one or more external inputs which partially determine a result.” Two kinds of parameters are distinguished in Lisp: variables in expressions whose value is determined by an environment, and function arguments whose value is determined at the point where a function is called. This corresponds to \textit{lexical parameters} resp. \textit{data parameters} in Lamping’s system. Lexical parameters are lexically scoped variables notated by variable names, whereas data parameters are associated with data objects and are evaluated in the location where the object is used. Since data parameters resemble variables in other respects “all parametrisation in the system is handled uniformly.”

An example of data parametrisation is the following: \textit{f} may be defined as

\[
\text{let } f \leftarrow \text{data } x : (x * x)
\]

and used in the following expression:

\[
(supply x \leftarrow 3 \text{ to } f) + (supply x \leftarrow 4 \text{ to } f)
\]
The *supply* construction is like a *let* construction but is used for data parameters. The construction *data* $x$ "indicates a potential dependency on the data parameter named $x$." Exactly as in the case of parametrised modules in Pebble (Section 6.2.6), data parameters define the communication capabilities of the data object where it is defined, and are thus not alpha-convertible. Data parameters are transparent, which allows parametrised objects to be used as if they were non-parametrised. This also allows the system to "generalise some of the parameter passing capabilities of dynamic binding." For instance, the following expression is possible:

$$
\text{let } f \leftarrow \text{data } x : (x * x + x + 1) \text{in}
\text{let } g \leftarrow f * f \text{ in}
\text{supply } x \leftarrow 5 \text{ to } g
$$

This expression calculates $(x^2 + x + 1)^2$ at $x = 5$. In the second line we may observe how $f$ is manipulated as if it were a ground object.

An environment is represented indirectly as a set of bindings, which can also be parametrised and thus augmented to affect some of the values it specifies. The parametrisation properties of object method lookup can also be expressed as a set of bindings. A module may also be represented as a parametrised set of bindings that by its turn is parametrised on the imports of the module.

In contrast to Symmetric Lisp or Pebble, discussed in the two previous sections, where sets of bindings are an additional language feature and first-class objects, in Lamping’s system a set of bindings is a consequence of parametrisation over specifications. Furthermore, in the former languages we cannot augment a set of bindings nor combine two of them.

**Quasi-Static Scoping**

Another attempt to come to terms with the issue of naming as a communication capability is given in [93], this time by the introduction of a new scoping discipline called quasi-static scoping. The authors describe it as a mechanism for runtime linking. Static scoping is presented as a “a strong encapsulation mechanism for hiding the details of program units” whose most basic limitation is that “it does not allow the sharing of variable bindings (locations) across independent program units.” In other words, no communication, or “cross references”, between independent program units is possible in the absence of dynamic scoping: “facilities such as module and object systems that require cross references of variables therefore must be added as special features.” Quasi-static scoping solves this problem by giving the user the power to decide the time and scope in which to resolve a quasi-static variable.

A resolution operation to resolve a variable is provided that may be performed independently of the variable’s evaluation time at the entire discretion of the user. The term quasi-static refers to the fact that once resolved, a variable’s reference remains fixed. What we have here is a clear example of the composition-restriction mechanism common in modular development of programs.

As long as variables are not resolved, their names are not alpha-convertible. Hence, “the name-based resolution of a quasi-static variable... does not survive alpha-conversion, since the scope of a shared quasi-static variable cannot be determined lexically.” The authors solve this problem by letting variables have two names, one internal and alpha-convertible, and other external and not subject to renaming.

In Scheme extended with quasi-static scoping, quasi-static variables are defined in *quasi-static procedures* as follows, in analogy to lambda abstractions:
The names $q_1, \ldots, q_k$ are the internal names of the procedure's quasi-static parameters, subject to alpha-conversion, whereas $Q_1, \ldots, Q_k$ are the external names. Resolution is performed by a special form. For details we refer to [93].

With the help of quasi-static scoping a module system and an object system are built. For modules, quasi-static scoping is used for variable sharing among modules. Modules communicate with each other by importing and exporting variables. To this end, two kinds of first-class entities are defined, interfaces and clients. The former defines a set of bindings whose external names are seen as sharable communication channels between clients, and whose internal names denote functions. A client is a quasi-static procedure that "imports" the external names of the interface via its quasi-static formals. Clients may be thus separately compiled for later linking with interfaces.

A system of objects is also described, based on message passing, single inheritance, self-reference via self, and subclassing via delegation. Quasi-static variables are used for code sharing among objects. The object's methods are defined independently of the object's instance variables by letting the method's quasi static formal parameters represent the instance variables accessed by the method. The resulting system is functional; the names of the methods are defined as symbols and assumed to be public. The use of quasi-static variables is thus reduced to the names of the object's instance variables. Otherwise, the treatment of objects is quite conventional, with message dispatchers and lists of bindings from method names to method definitions.

In contrast to Lamping's unified system of parametrisation (see Section 6.2.6), which uses the runtime dynamic environment to resolve the non-lexical variables, the quasi-static variables are resolved through an explicit resolution operation, which the authors claim make programs easier to analyse and to implement. Reasons of efficiency rather than semantics are also put forward in favour of the use of quasi-static variables instead of first-class environments, e.g. that in the absence of alpha-conversion macro systems could run into problems due to inadvertent capture of generated variable names.

First-class Extents

In Lee and Friedman [92], the notion of first-class extents is proposed instead of first-class environments, with the goal of preserving the lexical scoping discipline. According to the authors, first-class environments rely on the names of the variables, or identifiers, and thus do not mesh well with static scoping. In contrast to environments, first-class extents are defined on variables instead of identifiers, thus avoiding the name capture problems associated with first-class environments.

Variables are distinguished from identifiers. First-class extents rely on the semantics of variables. A variable with name $i$ is mapped to a value $v$ using a composition of two maps: $\mu$, binding the identifier $i$ to a variable $a$, and $\epsilon$, binding $a$ to $v$. The map $\epsilon$ is called an extent. First-class extents allow the definition of variables with lexical scope and dynamic extent, are not subject to name capturing and may be alpha-converted. These features may be described in terms of the new operator of the $\lambda$-calculus and some functional languages.

There can be multiple extents at any point during a computation, ordered according to the time of creation, the most recent one first. Variables refer to the most recent extent where they are bound, the current binding, and new bindings may be added to
the extents. There are operations that “unshadows” shadowed variables by allowing a previous current binding to become the current binding.

An object system is also defined with the help of first-class extents, which according to the authors lead to an “unusual approach to object-oriented programming.” In this system, there is no difference between classes and instances, both being defined as objects. Objects are viewed as a collection of attributes, e.g., class variables, instance variables, and methods. Attributes are denoted by lexical variables rather than identifiers, and are represented as extent bindings. An object is represented as a list of extents \((E_1E_2\ldots E_n)\), where the tail of the list \((E_2\ldots E_n)\) is the superobject. Messages are viewed as pieces of program and message sending as evaluation of the program within the extent of the receiver. Messages always refer to the latest binding in the receiving object. Self-reference is thus achieved “without resorting to any explicit mechanism such as the pseudo variable self”. Attributes of the receiving object may be accessed by ordinary variable reference. In this way modularity is enhanced and the security of lexically bound variables is preserved.

An interesting observation is made in [92] with respect to the relation between object-oriented programming and functional programming in languages such as Scheme. Ordinary Scheme programming is regarded as “a special case of object-oriented programming where computation is always carried out within the state of the Scheme object” which is viewed as an object whose list of extents is void. The system obtained by extending Scheme with first-class extents is seen as a “natural extension of Scheme.”

The authors distinguish between the representation of an object as a dispatcher and as an environment wherein some form of computation may take place: “a dispatcher is not a true environment.” Therefore “an object’s attributes are accessed through procedure invocations” rather than by ordinary variable references. This reflects in our opinion a functional bias that views objects as functions from messages to values, rather than as processes evolving within a determined environment. Thus, access of the attributes of an object is done via a message sent to the object along the pseudo-variable self, not as an ordinary variable reference. A problem with the dispatcher approach is the fact that “it cannot use lexical scoping to control the accessibility of an object’s attributes”, and must instead “rely on the uniqueness of the dispatcher’s domain… to hide its attributes from the rest of the program.” In the semantics of parallel object-oriented programming languages given in [180] by translation to the \(\pi\)-calculus, this uniqueness is achieved by the restriction operator. This implies that the public names of objects are not used as channels but rather as a kind of control mechanism for dispatching to the corresponding method, which is usually represented by an internal name that must be distinct for each existing object. Thus, even in process-oriented languages there has been a tendency to encode objects as dispatchers, in the functional style.

In contrast to quasi-static scoping, discussed in the previous section, first-class extents are not suitable for modularisation. The problem, not surprisingly, is related to the treatment given to identifiers in first-class extents, and makes itself felt in situations when “the system is incapable of figuring out the variable associated with an identifier.” As pointed out by the authors, “program module interfaces, the interconnections between separately compiled program units, must rely on some protocol that ultimately must be expressed at the symbolic level,” and “the identifiers of first class environments best serve this purpose.” In other words, the identifiers themselves, at the “symbolic level”, i.e., as names, must ultimately play a role in the definition of protocols that support interconnection and communication between separate units or modules.
Reification and Reflection: Rascal

In many languages, environments can be reified into concrete data structures (e.g. *Lisp-3* [157] and *Brown* [49]), and, conversely, data structures can be reflected into environments. Contexts in *Smalltalk* [56] and *Self* [171] may be also viewed as reified environments.

*Rascal* [72] is a language that permits semantic environments to be reified into data objects, and conversely to transform data objects into environments. Additionally, it allows users to constrain the extent and scope of reification and reflection. Such constraints are useful since the reification of environments may violate lexical abstraction and locality by unwanted capture of bindings, and reflection may impede useful program transformations that rely on static scoping.

The main purpose of Rascal is to provide different forms of dynamic binding through abstractions that allow the selective import and export of bindings. Lexical binding, as noted in [72], does not provide mechanisms for the import and export of bindings from and to different lexical contexts. According to the author, “object, modules, or incrementally constructed programs are several examples where such functionality is important.” Variable references are usually defined “via an abstract environment”, but “in most languages, this abstract environment is hidden from the programmer.” To remedy this, first-class environments are provided in Rascal. These are partitioned into two components, a lexical environment that contains lexical bindings exclusively, and a public environment that contains public bindings. The latter are bindings that are “exportable outside the lexical environment in which they are created” (cf. the notion of extrusion in the π-calculus), which is done by an operation called *make-public*. It is important to note here that what is made public is rather the binding of a previously public name to a value. An expression evaluated in such environment must “know” in advance the names bound in the environment in order to access them.

By reflection, data objects binding names to values may be transformed into a restricted scope within which expressions can evaluate. This is done by a *reflect* operation The set of bindings captured by a reification operation may be curtailed by a *barrier* operation, as well as the scope of expressions evaluated within the context of a reflected data object by a *restrict* operation.

Static environments are used to encode records, modules, and stacks. Dynamic environments, i.e. environments which are incrementally created, are used to encode a language’s front-end. It is also shown that in the presence of operations that explicitly capture an environment the lexical binding rule is “logically unnecessary,” and late-binding semantics can be ascribed to procedures or individual variables “without necessitating any alteration to the base language semantics.” As we see the inescapable notion here is once again that of environment and lookup operations into environments, rather than the binding discipline.

Operations provided by object-oriented constructs are also related to operations in Rascal by the author, who considers object-oriented computing “an important domain for environment metaprogramming.” Basically, objects are environments where instance variables and methods are public names. Inheritance is obtained by allowing reflection and reification to create modified versions of objects where old definitions are still accessible through projection and capture of environments. Classes are viewed as namespace generators, and a class hierarchy is built by composition of old and new instances of namespaces generated from the superclasses. The notion of *self* “is implemented using namespace composition, reflection, and reification,” not by recursion or fixed-point constructions. The pseudovariables *self* and *super* may be thus treated “as ordinary user-defined objects using operations
Label-Selective $\lambda$-calculus

In [51] another extension of the $\lambda$-calculus, called the label-selective $\lambda$-calculus, is introduced. In this calculus, the arguments of functions are selected by labels that may involve numeric positions as well as symbolic keywords. The latter enjoys “free communication”, according to the authors, while the former “must comply with relative precedence in order to preserve currying.” The distinction between references and $\lambda$ abstractions are blurred out. The extension is conservative since it coincides with the $\lambda$-calculus when the set of labels is reduced to the singleton $\{1\}$. Although the main reason for introducing labels is to support out-of-order partial applications, it has wider consequences, e.g. we get a syntax that is closer to the object-oriented style.

Symbolic keywords are used to specify arguments of functions in many programming languages, but they are seen as syntactic sugar and are usually “compiled away” as numeric positions. However, in languages that support currying this is not easy to do. The consumption of the arguments usually follow a strict left-to-right order. This is possible because of the natural isomorphism

$$A \times B \rightarrow C \simeq A \rightarrow (B \rightarrow C).$$

Now, since $A \times B \simeq B \times A$, we should also get

$$A \rightarrow (B \rightarrow C) \simeq B \rightarrow (A \rightarrow C).$$

The authors propose the use of the category-theoretic definition of the Cartesian product in which projections $\pi_1$ and $\pi_2$ are used explicitly, resulting in the expression $(\pi_1 \Rightarrow A) \times (\pi_2 \Rightarrow B)$ rather than simply $A \times B$. Hence, we get immediately

$$(\pi_1 \Rightarrow A) \times (\pi_2 \Rightarrow B) \simeq (\pi_2 \Rightarrow B) \times (\pi_1 \Rightarrow A),$$

and thus

$$(\pi_1 \Rightarrow A) \rightarrow ((\pi_2 \Rightarrow B) \rightarrow C) \simeq (\pi_2 \Rightarrow B) \rightarrow ((\pi_1 \Rightarrow A) \rightarrow C).$$

We obtain here immediately the desired permutativity of currying and a greater flexibility in defining the order of communication of the arguments of a function. The price paid for this, which is also the price usually paid for the benefits of dynamic binding, is the loss of “implicit argument positions as numeric offset,” analogous to the compilation of variables as references to a fixed location.

The syntax of a version of the selective $\lambda$-calculus with only symbolic labels is as follows:

$$M ::= \ x \ \mid \lambda_{\alpha} x. M \ \mid M_p M'$$

$M$ represents terms; $p$ denotes an element of an ordered set $L$ of labels; $x$ is a variable; $\lambda_{\alpha} x. M$ denotes the variable $x$ abstracted on $p$ in $M$, and $M_p M'$ represents $M$ applied on $p$ to $M'$. Abstraction and application of the same label form redexes. The rewriting rules for the selective $\lambda$-calculus make this clear:
\[ \beta - \text{reduction} \]

\[ (\beta) \ (\lambda p x. M)_{p}^{\gamma} N \rightarrow [x/\text{backslash}N] M \]

Reordering

(1) \( \lambda p x. \lambda y. M \rightarrow \lambda y. \lambda p x. M, \quad p > q \)
(2) \( M_{p}^{\gamma} N_{q} L \rightarrow M_{q}^{\gamma} L_{p} N, \quad p > q \)
(3) \( (\lambda p x. M)_{q}^{\gamma} N \rightarrow \lambda p x. (M_{p}^{\gamma} N), \quad p \neq q, x \notin FV(N) \)

Rules 1-3, the reordering rules, enforce termination of reordering. FV stands for free variables.

The authors consider that “an intuitive, but accurate, explanation of the label-selective \( \lambda \)-calculus can be given as extracting implicit concurrency from \( \lambda \)-calculus.” The idea is that there is an inherent concurrency in the \( \lambda \)-calculus that is lost because of the non-commutativity of function application. It seems that only in the presence of names or labels it is possible to introduce concurrency in the calculus. The authors consider that “a useful way of thinking of these symbols”, i.e the labels, “is to see them as channel names used for process communication.” A process is thus a \( \lambda \)-term, “where sending is performed by applications and receiving by abstractors.” In fact, the authors declare that “the intuition behind selective \( \lambda \)-calculus are no longer functions but functions over labelled arguments which behave like communicating processes through named or (relatively) numbered channels”, and “application corresponds to process communication,” all of which “starts to be strongly reminiscent of a calculus for process communication.” What connects the two calculi is the treatment given to names, which are seen as channels. However, the notion of environment is missing in both calculi. To remedy this, the transformation calculus was introduced, which is the subject of the next section.

The Transformation Calculus

The transformation calculus of Garrigue [50] is an extension of the label-selective \( \lambda \)-calculus with a notion of state and of composition that is compatible with currying. Currying is applied to the output of functions via labels on the output; and records and state are represented explicitly in the transformation calculus, in contrast to the label-selective \( \lambda \)-calculus. The transformation calculus is a conservative extension of the \( \lambda \)-calculus that provides a way of representing state transformations, a state being represented by labelled input parameters that may be returned by a term. In this way, a state can be extended dynamically, and parts of it may be ignored by a term.

The purpose of the transformation calculus is to provide a tool for a functional representation of stateful objects and to express state transformations by using labels as identifiers [50] [52]. Objects are represented as a collection of methods that are seen as transformations, and a labelled state. A method modifies the state of the object by a transformation modifying the value attached to some label belonging to the object. In this approach, methods are not meant to be invoked “by messages sent to the object,” but seen as “a set of transformations implementing methods.” The representation of methods and the state of the object are separate, a distinction which “avoids self-references,” as claimed by the author. We see here another example of the rather contingent significance of the notions of recursion and self-reference in relation to objects.
Records may be represented in the label-selective $\lambda$-calculus by the following expression:

$$\lambda_{\text{sel}}x.(x_{\ell_1}^\ell M_1 \ldots x_{\ell_n}^\ell M_n)$$

where the label $\text{sel}$ indicates that the expression should be applied to a selector, which is a function that returns the value associated with the desired field and discards the rest. The expression

$$\lambda_{\ell_1}y \ldots \lambda_{\ell_k}x \ldots \lambda_{\ell_n}y.x$$

selects the field $l_k$ and discards the other fields. A record may be represented by the expression

$$l_1^\ell M_1 \ldots l_n^\ell M_n$$

which is not a term. Basically, we get the transformation calculus by incorporating this expression into the syntax of the label-selective $\lambda$-calculus. For details we refer to [52].

In [50] the claim is made that the transformation calculus permits both functional and imperative encodings of algorithms, that it shows the relation between the $\lambda$-calculus and algorithms, and that it gives a “basis for functional languages handling states and sequentiality problems.” This supports the view that computation is basically a sequence of transformations performed on a collection of environments, a fact which extends also to pure functional languages where environments are usually not represented explicitly.

The system of labels is viewed as an important asset for the transformation calculus in [50]. It is also suggested that the calculus might be extended to allow for the generation of new label names to give “unique identifiers for scope-free variables,” in the same spirit as the restriction operator in the $\pi$-calculus, thus avoiding “the hiding of a label in those subsequences which create new variables on this label.” It is also claimed that “a more structured label space even enables the use of object-based techniques, and solve the restrictions of dynamic binding.” We see here a clear example of the interplay between name hiding and fresh name creation, and the roles these notions may play in the structuring of environments or states.

Transformations can be composed in a way that describes changes on a mutable state combined with functional computations. According to Garrigue [53]

> The notion of mutable state is essential in many algorithms. There are ways to turn it, with recursion and infinite lists, but this is not always very natural. A more intuitive notion of state is helpful; we can see it even in mathematics, where algorithms are often defined imperatively. Lambda-calculus is enough to define functions, but not always practical for algorithms.

**Lambda Calculus with Contexts**

There have been several attempts to formalise the notion of context in the $\lambda$-calculus, defined as “holes” in lambda terms. The reason for choosing this framework is not altogether clear, since the $\lambda$-calculus, with its strict scoping mechanism, does not seem to be the most adequate formalism for the study of contexts. Hence, this line of research has been focused on issues that are inherent of the calculus itself, such as alpha-convertibility or commutative of hole-filling and renaming, and less on the meaning of name binding for communication.
 Apparently, the first effort in this area, often neglected in later works, is due to Fraser [11]. In this work, a formalism is developed to describe context manipulation that contains the key ingredients that are to appear in later works. Fraser extended Church’s λ notation to deal with contexts by swapping the terms of a function application. Whereas a function can be described as an abstraction of type \( \lambda(namelist)(expression)(valuelist) \), a context is viewed as an abstraction of type \( \mu(namelist)(valuelist)(expression) \).

In [94] Lee and Friedman propose a calculus where two disjoint variable name spaces are assumed, one for lambda terms and other for contexts, which are also two disjoint classes of terms. A context is defined as a term in the \( \lambda \)-calculus with holes whose filling enables name capture, in contrast to \( \beta \)-substitution. This “seemingly simple feature,” which is “at odds with hygienic \( \beta \)-substitution” ... transcends static scope and lies as the heart of modular and object-oriented programming,” as claimed by the authors. The result is a system in which \( \beta \)-reduction and fill-reduction, i.e., hole filling, are two separate operations.

The \( \lambda \)-calculus is conservatively extended in order to incorporate contexts without jeopardising the \( \beta \)-rule, yielding the \( \lambda C \)-calculus. Contexts are viewed as source code, \( \lambda \)-terms as target code, and context-filling as compilation operations. Free import and export variables are renamed during the linking of separately-developed modules, which are regarded as first-class objects. To this end, parameters are introduced that are based on names which are not \( \alpha \)-convertible and which are linked dynamically. Module linking is modeled by the renaming of import and export free variables. Relinkable variable references are used to model virtual method references in order to define object systems. According to the authors “the inclusion of contexts introduces parameters whose linking is based on names (symbols, identifiers, or keywords).”

There is thus a strict separation between terms and contexts. Terms are those of the \( \lambda \)-calculus. A term \( e \) is either a variable \( x \), an abstraction \( \lambda x.e \), or an application \( ee' \). Contexts, denoted \( C \), on the other hand, are defined inductively as an identifier \( x \), an abstraction \( \lambda x.C \), an application \( CC' \), or a hole \( h \). Contexts and terms share thus the same syntactic structures but are “based on distinct categories of names: variables \( x \) versus identifiers \( x \) and holes \( h \).” For terms, \( \beta \)-substitution is defined as in the \( \lambda \)-calculus, but hole filling \( [C/h]C' \) is defined as syntactic substitution, with possible capture of free variables in \( C \). Contexts may be viewed as “a keyword parameter-passing mechanism when identifiers are seen as keywords.” Contexts are thus parametrised on the keywords that occur in it as free identifiers. If \( x \) occurs free in the context \( C_2 \), the substitution of \( C_2 \) for \( h \) in let \( \{ x = C_1 \} \) in \( h \) results in let \( \{ x = C_1 \} \) in \( C_2 \), associating any free occurrences of the identifier \( x \) in \( C_2 \) with the context \( C_1 \). Hole-filling may be seen as integration of modules, names or identifiers as a communication facility enabling separate development of program modules, and contexts as program modules that export through holes and import via free identifiers. Names are used as “a common domain to communicate among program modules,” not simply as identifiers or variable names. It is explicitly recognised that “names are essential to programming large flexible software systems.”

Contexts without holes are fully evolved source code that can be compiled and linked. Computation is nevertheless modelled with \( \beta \)-reduction. Mechanisms to construct and compile source code, as well as to link and execute compiled code, are also defined. Briefly, these mechanisms are the following: abstractions to simulate compiled evolved contexts; an operation to execute compiled code; an operation to build compiled \( \lambda \)-abstractions; and an operation to build compiled operation code. For details we refer to [94].
Relinkable variable references are also added to the calculus in order to account
for object-oriented features. Related name-based mechanisms such as records,
the \( \lambda \)-calculus with names (see Section 6.2.5), the label-selective \( \lambda \)-calculus (sec-
tion 6.2.6), transparent data parameters (Section 6.2.6), and quasi-static procedures
(Section 6.2.6 ) are also discussed.

A related work is [82], where Kahrs proposes a combinatory term rewriting system
compatible with contexts, but in which contexts and hole-filling are not explicitly
represented as terms. Kahrs addresses the problem of the incompatibility of contexts
with \( \alpha \)-congruence and \( \beta \)-reduction, an incompatibility which we believe is related
to the communication aspects associated with the operation of hole-filling and which
needs not be considered a drawback.

Talcott [165] presents an algebraic system providing support for representing bind-
ing mechanisms and structures with holes. The work targets symbolic structures,
which embody notions of binding and require rules with hygienic side-conditions to
avoid name conflicts. Context filling is described as a mechanism for capturing free
variables. Binding structures provide for named free variables, nameless bound
variables, and holes. The notion of binding context is introduced to account for
externally bound variables. According to Talcott, the notion of hole appeared first
as “a means of representing symbolic computation.” Nevertheless, holes belongs
to the meta level system, and neither contexts nor hole-filling is represented in the
reduction system of the \( \lambda \)-calculus.

Substitutions in the \( \lambda \)-calculus appear as meta-level objects. In the \( \lambda \sigma \) calculus [2]
they are made explicit at the theory level in order to allow implementation of the
\( \beta \)-reduction. Metavariables that behaves like holes are introduced, but not treated
further.

Another relevant work in this context is Hashimoto and Ohori [63], where a typed
calculus for contexts is presented. The calculus contains also labelled holes, hole ab-
straction and context application for manipulating first class contexts. The purpose
of the calculus is to support the development of a programming language incorpo-
rating the manipulation of open terms, whose behaviour depends on the bindings of
their free variables. Contexts are first-class objects. The authors claim that this fea-
ture, apart from providing a flexible program environment and first-class modules,
also supports program migration in distributed programming. Open terms may be
transmitted to a remote site where the necessary binding is executed, instead of
transmission of the entire closure which would be very inefficient. According to
the authors, “a typed calculus with first-class contexts would provide a clean and
safe mechanism for manipulating open terms.” A typed calculus for contexts is
developed where contexts and lambda terms belong to the same syntactic category.
The authors suggest that it is possible to establish a calculus where substitution
and hole-filling are defined on the same set of terms. The resulting calculus is
Church-Rosser and the type system has the subject reduction property.

In order to allow \( \alpha \)- convertibility of bound variables in the presence of contexts,
Pitts [137] introduces function variables to represent holes, and substitution is done
by substituting meta-abstractions for these function variables. This is developed fur-
ther by Sands [147]. It should be pointed out that the method of meta-abstractions
boils down to the use of abstraction over originally free names in a term, as ex-
plained in Section 1.4. Since those names are abstracted, they must be instantiated
by application, not by dynamic binding, which enables alpha-convertibility.

For the sake of completeness, we mention briefly the following works. Michel [104]
addresses the problem of name and name capture by considering expressions with
free names as incomplete expressions which are dynamically completed by a name
capture mechanism. The formalism allows the introduction of first-class environments, and is applied to model distributed incremental program construction and to define an object-oriented programming style. Another calculus with environments as first-class values, called \( \lambda e \), was introduced by Sato et al. [151]. In this calculus, environments are regarded as a generalisation of explicit substitutions and records, and can be computed by function application and evaluation in other environments. Sato et al. [152] introduce another variant, the \( \lambda e \) calculus, where contexts are represented by ordinary variables and hole-filling by functional application together with a new abstraction mechanism for packing and unpacking terms. Finally, Bognar and de Vrijer [24] developed the \( \lambda c \) calculus, a general framework for representing contexts which can be manipulated before or after hole filling by a mechanism of delayed substitution.

A Syntactic Theory of Dynamic Binding

In [116], a syntactic theory is proposed that supports equational reasoning on programs using dynamic binding. It uses a translation from a notation using dynamic binding, the language \( \Lambda_u \), to one using only lexical binding, the language \( \text{deps}(\Lambda_d) \), based on the dynamic environment-passing translation, which adds to each function an extra argument representing the dynamic environment. References to dynamic variables are translated into a lookup operation in the current dynamic environment.

The language \( \Lambda_u \) is based on two disjoint sets of variables, the dynamic and static variables. There are two types of lexical abstraction, \( \lambda x_s. M \) for abstracting lexical variables, and \( \lambda x_d. M \) for abstracting dynamic variables. In other aspects \( \Lambda_u \) it is similar to the \( \lambda \)-calculus.

The target language \( \text{deps}(\Lambda_d) \) is an extended call-by-value \( \lambda \)-calculus with only lexical variables, but instead of a set of dynamic variables we have a set of constants, which may be described as a set of identifiers or names. Special terms are defined for looking up the values of constants. The term \( \text{lookup} \hat{x} E \), where \( \hat{x} \) denotes a constant and \( E \) an environment, denotes an abstraction that takes an environment as one of the arguments; the term \( \lambda(\hat{e},y).P \) denotes an application involving the previous type of abstraction; and finally the \( \text{extend} E \hat{x} W \), where \( W \) denotes a value, is used for extending environments.

The translation maps each abstraction, dynamic or static, into an abstraction taking an environment in the first argument. Dynamic abstractions are translated into abstractions which extend the environment. Dynamic variables are represented by constants, and are translated into a lookup for the corresponding constant in the current dynamic environment. For the purposes of our work, the point that must be highlighted here is the fact that what are called constants in the \( \Lambda_u \) calculus are basically free identifiers or in other words names, which are consequently unbound and are used for communication purposes.

The authors also prove that dynamic binding adds expressiveness to a purely functional language. This is done using Felleisen’s definition of expressiveness in [45].

Mutable Variables

Here we review briefly a series of related formalisms and languages that have some relevance for our subject. In contrast to the formalisms we have discussed in some detail above, the ones discussed here do not explicitly incorporate the notions of names, labels, records or states, since their purpose is mainly to preserve referential transparency in the presence of assignment operations.
In [75] and [76], stores and partial continuations are presented as first-class objects in the programming language GL. In GL stores are assignable as values of variables and “subject to other similar manipulations,” which implies that “several different versions of ‘computer memory’ during a program’s execution may exist in the interactive program environment simultaneously.” In other words, environments may be partitioned and “reflexively” made explicit at the language level, a capability that “turns out to be quite useful in an environment for purposes of interactive program testing.”

In [178] Wadler presents a pure functional programming style as an alternative to functional programming with assignments. This is done by hiding the state through monads [177]. Although a centralised definition of state is required, monads eliminate explicit mention of state arguments. As Wadler observes, the method “does not completely eliminate the tension between purity and impurity, but it does relax it a little bit.” The notion of monad comes from category theory, and the technique used by Wadler was later developed by Eugenio Moggi [114] with the purpose of giving a semantic description of such notions such state, exceptions and continuations. In [80] monads are used to add interaction to a pure functional language. In [58] and [38] similar models are proposed.

Monads have also been used as a basis for adding state and concurrency to a functional language [81]. In Concurrent Haskell, concurrency is integrated into a lazy purely-functional language via addition of processes and atomically-mutable states to support inter-process communication and cooperation.

In [163] and [164] a theoretical framework is presented for adding mutable references, assignments and dynamic data to functional languages that do not violate their semantical properties, i.e. strong normalisation, confluence and referential transparency. A calculus called ICL, the imperative \( \lambda \)-calculus, is presented. It is based on the principle that inspection, or observation, is the only manipulation needed for states, which may be implicitly extended and observed, via use of variables, but never explicitly manipulated. As the author put it, “the world exists only to be observed.” Two kinds of variables are defined, conventional variables and reference variables, the difference being that distinct reference variables always denote distinct references in any term, in contrast to conventional variables. A type system is also provided that distinguishes between state-dependent and state-independent expressions.

Another related formalism, \( \lambda_{\text{var}} \), is presented in [132], where it is described as an extension of the call-by-name \( \lambda \)-calculus with constructs representing mutable variables and assignments. There is no explicit notion of store, and state is represented by the collection of assignments in a term. It is claimed that the calculus provides a framework combining functions and state “in a way that can naturally express advanced imperative constructs without destroying the algebraic properties of the functional subset.”

6.3 The Object Paradigm

Interaction and state are essential elements of object-based systems. Nevertheless, functional models of computation cannot adequately account for these phenomena. In the concurrent paradigm, objects can be viewed as processes, method invocation as a kind of communication, and encapsulation as restriction of the visibility of operators. However, in the \( \pi \)-calculus there are no primitives for scope restriction in this sense, since the restriction operator in the \( \pi \)-calculus only declares a name as globally unique.
As we argued in the preceding section, referencing is pervasive also in the functional model of computation. However, the absence of references in the \( \lambda \)-calculus, the theoretical foundation for the functional paradigm, seems to have contributed to relegate referencing in computation to the role of a convenient practical computing device without deeper theoretical significance. The question is whether such computationally significant features as referencing should be allowed to remain hidden in models of computation, or should rather be given a firm theoretical status in their own right as essential elements of the model. An example is the notion substitution in functional languages. Substitution was regarded as belonging to the informal meta-level description of the \( \lambda \)-calculus, not to the notation itself. But computationally substitution is an essential ingredient, and to close the gap between theory and practice the notion has been lately explicitly integrated in e.g. the calculus outlined in [2].

An explicit theory for naming and referencing emerged with the advent of the \( \pi \)-calculus, in which it is the key idea. But in an incipient form it is already present in the characterisation of processes as transducers, which is the subject of the next section.

The rest of this section is organised as follows. In Section 6.3.1 we present in detail an early paper of Milner on communication and interaction, and focus on the role played by names in this work. Section 6.3.2 is dedicated to one of the most influential models of concurrent computation, the actors model. In Section 6.3.3 we present and discuss several encodings of objects in distinct process algebraic formalism, where the difficulties occasioned by the lack of dynamic binding primitives are brought forward.

6.3.1 Transducers

In [110] Robin Milner argued that “most of the computing agents with which computing science is concerned... exhibit a behaviour which is not just the computation of a mathematical function of their inputs, but rather a possibly infinite sequence of communications with their environment.” These computing agents should be viewed as transducers from sequences of responses from the environment to sequences of responses to the environment. Furthermore, Milner explains that a “crucial feature is the ability to define the operation of binding together two processes to yield another process representing the composite of two computing agents, with their mutual communications internalised.” This is basically the notion of agent compositionality as composition plus restriction, analogous to the way two electronic components are plugged together. However, what we see emerging in the transducer model is the novel role given to labels, i.e. names, as a kind of communication channel.

According to Milner, it is necessary to “capture the side-effect history of a program in its meaning”. “Side effect” was a rather derogatory term used by the advocates of applicative programming, as noted by Hewitt in [68]. In consequence, Hewitt preferred to use the term effect, which he defines as “a local state change in a shared actor which causes a change in behaviour that is visible to other actors.” We could thus define an effect as a change in the local or global environment, which is immediately visible to other agents. In sequential languages we commonly have a unique environment that is always active throughout a computation; by contrast, in concurrent languages there may be more than one that are active at the same time. At a deep level both applicative and imperative languages may be defined as “effect” languages, i.e. languages whose constructs may be defined as environment modifiers [161]. Thus, capturing the side-effect history of a program in its meaning amounts
to capturing the history of the effects caused on the environment by computation, i.e. the “memory-access history of programs” [110]. Moreover, “the meaning of a program should express its history of access to resources which are not private to it.” [110] This history of access to external resources defines the “overt behaviour” of the agent, and it “constitutes the whole meaning of any computing agent.” But resources that are not private must be necessarily referenced by a term which is not private, e.g. a public or shared name. It is not surprising that in this context the notion of name should find its way in a notation that gradually developed into CCS and later into the π-calculus, in which it is the basic notion. Names can be regarded as the “barbs” of a process, i.e. the visible interface that determine the communication capabilities of a communicating agent, and ultimately as the elements that define an agent’s overt behaviour.

We turn now to the formal definition of the transducer given by Milner. Let $L$ be the set of addresses used as labels the communication lines of the transducer, and $V$ the set of values. Agents are defined with the help of the automata-theoretic notion of transducers, formally a quadruple

$$\langle S, s_0, f, g \rangle$$

where $S$ is a set of states, $s_0$ the initial state,

$$f : S \rightarrow (V \rightarrow (L \times V))$$

the output function, and

$$g : S \rightarrow (V \rightarrow S)$$

the state transition function.

Processes are defined as a domain satisfying the recursive equation

$$P \cong V \rightarrow (L \times V \times P)$$

where the sets $L, V$ and $S$ are redefined as complete lattices by adjoining a top and a bottom element. Hence, a processes is a function that given a value in $V$ produces a pair $\langle \alpha, v \rangle$ as well as the “continuation” $p$, a new process. In this pair, $\alpha$ denotes a communication line or label and $v$ the value to be transmitted along $\alpha$. Note the “unconventional” use of the syntactic domain $L$ in the definition of a process (see Section 6.2.3), a reflection of the fact that labels are being used here for referencing rather than as variables denoting a value.

The behaviour of a transducer may thus be represented by a process $BEH(s_0)$, where $BEH : S \rightarrow P$ is defined recursively by

$$BEH s = \lambda u. (fsu, BEH(gsu)).$$

Let $C = (L \times V) \times V$ be the domain of communications, $t,t' \in L \times V \times P$, and $\gamma \in C^*$ be a communication sequence. The relation

$$t \xrightarrow{\gamma} t'$$

is defined by:

(i) If $\epsilon \in C^*$ is the null sequence, then $t \xrightarrow{\epsilon} t$. 

176
(ii) If \( t = (\alpha, v, p) \) and \( pu \xrightarrow{\gamma} t' \), then \( t' \xrightarrow{(\alpha, \gamma, v)} t'' \).

The domain \( V \) is assumed to contain a distinguished element \( \ominus \) denoting "no value", and the domain \( L \) a distinguished element \( \epsilon \) denoting a result address. Binding resources to addresses is done by defining an operation

\[
BIND \in L \to P \to P \to P
\]

Hence, \( BIND \ app' \) denotes the result of binding \( p' \) to \( \alpha \) in \( p \), or a process in which communications to \( \alpha \) in \( p \) are internalised and are served by \( p' \). We can interpret the occurrence of \( \alpha \) in a \( BIND \)-expression as a binding occurrence, and the final process as the result of applying the definition \( BIND \ app \in p \to P \) to a process, the resource. But this analogy seems a little contrived, since it implies an asymmetry between resource and resource users. To remedy this, a new domain \( Q \) is introduced satisfying the isomorphism

\[
Q \cong Q \to P
\]

and \( BIND : L \to P \to Q \) is now redefined as

\[
BINDapp = \lambda q . \lambda w . ((pv)_1 = \alpha) \quad \begin{cases} q(BIND\alpha(pw)_3)(pw)_2, \\ ((pv)_1, (pv)_2, BIND\alpha(pw)_3q). \end{cases}
\]

where \((pv)_i\) denotes the projection of \( pv \) to its \( i \)th element. In \( BIND \ app' \) the resource denoted by \( p' \) is internalised and only accessible to the the agent \( p \). Nevertheless, the dynamic aspects of process "application" are apparent in the way labels are constantly rebound. For instance, in the definition of \( BIND \) the label \( \alpha \) is rebound at every step to \((pv)_3\). Furthermore, the label is seen here as an address, not a process variable. Thus, two communicating processes in general are defined by

\[
p'' = (BIND \alpha p)(BIND \alpha' p'),
\]

which is explained as a process in which "control will pass back and forth between \( p \) and \( p' \) via the addresses \( \alpha \) and \( \alpha' \)." We may question whether \( \alpha \) and \( \alpha' \) are in fact bound in this context; we could also say that control passes back and forth between \( p \) and \( p' \) via fresh addresses every time it occurs. Another explanation would be that addresses \( \alpha \) and \( \alpha' \) are dynamically bound to new processes, the continuation of \( p' \) or \( p \), every time control is passed on.

In our opinion the semantics of names is already apparent, albeit in an incipient manner, in this early version of Milner’s communicating processes. It is further developed in [106] by allowing dynamic binding of addresses. Labels become action names, which in our opinion is a natural development of the notion of label used in the definition of a transducer. In this paper Milner declares that “it is important that behaviour describes possible activities of an agent. It is potential rather than actual, since what actually occurs will depend upon which communications are offered by the environment.” An agent is thus “open” in the sense that its meaning is dependent on the kind of environment in which it may take part. There is a close analogy here between a communicating agent and a function with free variables intended to be bound dynamically. This role is played in the case of communicating processes by the names that occur unrestricted in the agent. These names denote
the interface or communication capabilities of the agent and are intended to be bound dynamically within the environment in which the agent eventually evolves.

In [106], processes are assumed to possess ports along which they may communicate, and names and conames are introduced for labelling input resp. output ports. Input ports are associated with a function’s argument places, and output ports with their result place. “What is different is that a behaviour may input and output more (or less) than once - even infinitely often”, which resembles the “behaviour” of a function in an applicative environment. The difference is that now “it is also possible that inputs at the same port may have different sources, and outputs at the same port different destinations”. The resulting environment may be seen as a generalisation of the notion of environment used in applicative languages, in which the distinction between passive agents and active agents, i.e. between data structures and programs, is eliminated. Imperative programming constructs are not necessary, since an environment or state is represented by the agents themselves, which evolve by taking part in communication events.

In [8] Abramsky makes some criticisms concerning the definition of interaction in terms of names or labels. According to Abramsky “interaction becomes extrinsic” because some additional structure is required, e.g. a synchronisation algebra [186] “which implicitly refers to some external agency for matching up labels and generating communication events, rather than finding the meaning of interaction in the structure we already have.”

Another criticism is that “interaction becomes ad hoc”, and that many possibilities arise since it is an “invented” additional structure.

A third criticism concerns the lack of modular concepts in process algebra. Abramsky claims that “interactions becomes global” in Milner’s approach: “using names to match up communications implies some large space in which potential communications ‘swim’…” This is, in our opinion, a serious weakness in the \( \pi \)-calculus and related calculi, which is testified by the recent appearance of several related calculi which include primitives for encoding localities, e.g. the ambient calculus [33] or the Seal calculus [176]. The scope of names may be delimited in the \( \pi \)-calculus by the restriction operation. However, “the local character of particular interactions is not immediately apparent, and must be laboriously verified”, as pointed out by Abramsky [8]. In fact, the restriction operator in the \( \pi \)-calculus may be scarcely seen as a scoping operator, since it may be exported by extrusion, or moved by application of the structural rules. The restriction operator might be more adequately described as means of creating fresh names on a global scale. The lack of locality in interactions, according to Abramsky, “appears to account for many of the complications encountered in reasoning about concurrent object-oriented languages modelled in the \( \pi \)-calculus” [8]. One of the purposes of this thesis is to find primitives that mesh naturally with other primitives in the \( \pi \)-calculus and that might mitigate this.

Abramsky refers also to a couple of papers by Cliff Jones [79, 78]. In these works, Jones points out the difficulty of carrying out proofs about the semantics of concurrent object-oriented languages because interferences cannot be ruled out. For instance, as pointed out in [78], we cannot conclude that \( P \) and \( Q \) cannot affect each other simply because \( fn(P) \cap fn(Q) = \emptyset \), as illustrated by

\[
\pi y. y(z).0 \mid x'(y'').y''z'.0
\]

where the two terms in the composition might interact if composed with \( x(y').x'y' \), unless we block \( y''z' \) by \( y \) (or by \( y'' \)) in \( x'(y'').y''z'.0 \):

\[
\pi y. y(z).0 \mid x'(y'').(y''z'.0)y
\]
Jones suggests partitioning “the state in a way which shows that computation within a particular ‘island’ cannot affect any objects outside the islands” [79]. In the π-calculus, as we have seen, we cannot be sure that names are “uniquely handled”, according to the definition given in [79], i.e. that “names of unique objects never appear in object positions.” Jones suggests indexing process definitions “in a way which brings a notion similar to local states into the π-calculus” and the use of “the notion of islands as a way of dividing a large composition into sub-terms which do not interact with one another.” One way of achieving this, as we try to demonstrate in this thesis, is by blocking communication along selected channels.

6.3.2 Actors

The actor model [67] is a model of concurrent computation that emphasises the communication aspects of computation. Communication is defined in terms of message passing that resembles a mail service or datagrams. A message arrives at a computational agent called actor, the target of the message [36]. The arrival of a message is called an event. An event may activate subsequent events, or only produce an effect on the local state of its target. There is no notion of global state, which may be seen as being decomposed into local pieces. The overall computation can be described as a set of local computations interacting through message passing.

The main design decisions in the actor model concern the kind of objects required, the kind of messages each kind of object should receive, and the behaviour of each kind of actor upon receipt of a determined kind of message [67]. There is no “action at a distance” in the model; all computation is local.

In [68] Hewitt anticipated the increasing importance that applications based on independently developed communicating systems would acquire in the future. Many of his conclusions are highly relevant for our subject.

Hewitt regarded what he called the “closed world assumption” as contrary to the nature of open systems. The closed world assumption postulated that “all and only the relationships that can possibly hold among objects are those implied by the local information at hand.” By searching their local storage, systems would be able to acquire a complete picture of the world. This was clearly unrealistic, according to Hewitt, to whom concurrency in open systems “stems from the parallel operation of an incrementally growing number of multiple, independent, communicating agents.” Open systems are dynamic in the sense that their configuration are continuously evolving through the dynamic addition of new sites. Open systems are inherently concurrent, whereas the λ-calculus simulation is sequential. The latter may reduce or evaluate various parts of a single term or expression concurrently in a fixed environment. In contrast, a computation in an open system may be spawned incrementally by some independent agent and even affect the evaluation of expressions that started before the computation was initiated. Hewitt concludes that “the continuation technique promoted by Strachey and Milne [105] for simulating certain kinds of parallelism in the λ-calculus does not apply to Open Systems.”

A local state change in a shared actor is called an effect. A state change causes a change in the behaviour of the actor that is also visible to other actors. Hewitt refused to use the more usual term “side-effect” which he regarded was used pejoratively by advocates of purely applicative programming. Effects in the actor’s model are implemented “by an actor changing its own local state,” which is done via a become command, not by assignment. This model contrasts, as Hewitt observes, with other “models in which change is modeled by updating the state components of a universal global state,” as in [105]. This resembles the way that the notion of state is defined in process calculus formalisms, cf. the translation of an imperative
language into CCS given by Milner in [109], Chapter 8.

Clinger [36] showed that actor systems can perform nondeterministic computations that nondeterministic Turing Machines cannot perform. As a result, parallel systems cannot be modelled as nondeterministic sequential machines. Hewitt concludes that “recursive functions do no provide an adequate model of parallelism. I.e. an Open System cannot be adequately modeled as a recursive function which maps global states to global states because at any given point in time an Open System does not in general have a defined global state” [68].

The contrast between description and action, or describing and doing, according to Hewitt, is related to the contrast between truth and behaviour. Whereas the meaning of a sentence is the set of models which make it true, the meaning of a message in the message passing semantics is “the effect it has on the subsequent behaviour of the system.” The meaning of a message is thus “open ended and unfolds indefinitely far into the future as other recipients process the image.”

We could refine this distinction by distinguishing between doing and acting. The difference we have in mind is related to the difference between sequential and concurrent computation under the closed world assumption on the one side, and distributed computation in open systems on the other. Under the closed world assumption there is no uncertainty concerning the results of computation, although there may be non-determinism. In contrast, in an open system the result of an action is inherently uncertain and potentially infinite or non-terminating. More than just non-determinism is involved here, which implies that the set of results at each step could in principle be determined, which is captured by a denotational semantics based on powerdomains. However, in open systems we cannot always know in advance what will be the result of any action. For instance, in object-based systems we might never know what kind of new objects, classes or subclasses may be dynamically added to the system, and much less their behaviour. Any computational step taken in this environments may be rightly called an action, a term that suggests uncertainty in the outcome or final result of any performed step. A denotational semantics is here inappropriate, since it usually presupposes a complete or closed system.

Hewitt concludes that “description languages based on first order logic and/or the $\lambda$-calculus have been designed to express properties but are incapable of taking action... On the other hand procedural languages (such as current dialects of Lisp and Ada) have been designed to efficiently take action but they suffer from a lack of descriptive capabilities.” The actor’s model is an attempt to integrate the roles of description and actions.

### 6.3.3 Modelling Objects in Process Algebraic Formalisms

This section is dedicated to a survey and in-depth discussion of the various ways that objects have been modelled in process calculi, including CCS, Plain CHOCS, the $\pi$-calculus, and PICT, and some other related formalisms.

#### Modelling Objects in CCS

In [103] (panel discussion), Abramsky described Milner’s CCS [109] as “a notation for describing concurrent systems built up by uniform operators, combinators of communicating processes, and encapsulation,” and suggested applying CCS for the study of object-oriented features. He proposed viewing objects as processes that interact via method invocation, and suggested modelling encapsulation by restricting the visibility of operators.
Early examples of the approach of “semantics by translation” from concurrent programming languages into CCS [109] or other related process calculi, are [172], [129], [133] and [179].

Michael Papathomas outlines in [133] a framework for the description of concurrent object-oriented languages based on CCS. The approach is based on Milner’s translation of an imperative programming language in [109], extended to integrate object-oriented features.

The resulting system consists of a number of agents in CCS representing objects, classes and methods. Programs are translated into a parallel composition of agents denoting classes and a start-up agent representing the main program. Each class is represented by a request channel that accepts requests for creating new objects. Replies are represented by reply ports indexed by invocation identifiers, supplied by the caller.

Agents denoting objects consist of a composition of the several agents. A request handler accepts requests and interacts with the method agents for the execution of the method. This interaction is mediated by a method scheduler that controls the concurrent execution of methods. Variables are represented by instance variable agents as in [109]. A self agent is a special kind of variable that returns the object identifier. Method agents accept activation requests on internal restricted channels parametrised by the method name. Variables and method identifiers are part of the sort of the agents representing the object or the methods, and are restricted to ensure that their scope is localised to these agents. Self-reference is achieved through the special variable self, which is translated as an ordinary variable denoting the object identifier. No notion of recursion is involved here. An invocation of a local method translates simply into a request addressed to the request handler of the object sending the message.

The basic framework is also extended to support class inheritance. Overriding of methods is accomplished with the help of the restriction operator. The access sort for the method agents, including those in the superclass that are not overridden as well new or overridden methods, is restricted within the agent that encodes the object. The latter is nested within another agent denoting the superclass, which is a composition of the object agent and the method agents of the parent superclass. By repeating this procedure, an hierarchy of nested agents results, each one representing a superclass of the other. Calls to parent methods are done via events which generates a call to the parent method in the required scope. Since the agent that encodes the the object, identified by self, belongs to the most restricted scope of the hierarchy, the most specialised methods are always invoked, thus enforcing the discipline of late binding in method calls.

The limitations of this approach are basically those of the CCS formalism, i.e. a static interconnection structure and lack of support for dynamic creation and establishment of communication links. A more elegant solution is possible in the π-calculus, which supports these features.

Modelling Objects in Plain CHOCS

CHOCS [167] is an extension of Milner’s CCS with process passing, where the restriction operator is dynamic and not a scope binder. By contrast, Plain CHOCS adopts the static scope binding discipline for the restriction operator. Thomsen’s thesis [167] includes a study of the connection between concurrency and object-oriented programming. In this study, a small toy language O is defined and its semantics is determined by a translation into Plain CHOCS, which according to
Thomsen resembles a denotational semantics without the semantic domains. In \( O \), classes may be defined and objects instantiated. Each class is endowed with methods and a thread of control. Objects may execute in asynchronous parallel, synchronisation being obtained by method calls. A register is associated not only to each variable, but also to each class, object and method. The language was inspired by Milner’s \( P \) [107] and is untyped. It includes declarations, expressions, and commands. Declarations include variables, objects, methods and classes; expressions consist of variables and function calls; commands include assignment, sequencing, choice, while-loops, block, object creation, object initiation and method call.

The semantics of \( O \) is viewed as “a set of derived operators in Plain CHOCS.” In order to allow other values than processes in Plain CHOCS, Milner’s technique [108] of introducing \( D \)-indexed families of actions is adopted.

This pioneering attempt at giving a semantics by translation to an object-oriented language illustrates many of the limitations that are due to the absence of dynamic features and channel passing in the target language. The lack of dynamic name passing and the consequent need to treat classes, objects and methods as values for channel indexing yield an awkward object representation. The treatment given to method names and method calls is unclear. Apparently, the creation of objects result in the renaming of class methods, for purposes of indexing, to names that are unique for each object. This implies that the object that sends a message with an invocation of a method in the receiving object must know in advance the corresponding method names. Since there is no communication of names in Plain CHOCS, this assumes that the whole structure of the system must be predetermined. Objects must be declared in the declaration part of the block, and registers must be determined for each object, class or method in the system. Dynamic object creation is consequently not possible, and the overall structure is essentially static, with registers being allocated, as it were, at compile time.

**Modelling Objects in the \( \pi \)-calculus**

The \( \pi \)-calculus has the ability of expressing the dynamic creation of ports and their communication among agents. Walker takes advantage of these features in [179] to model a pair of parallel object-oriented languages with differing communication mechanisms, whose encodings in the \( \pi \)-calculus “are very close to natural operational intuitions.”

The translations for a couple of languages are put forward. Entities of distinct syntactic categories are represented as agents. The translations of declarations, expressions and statements are similar to those presented in [109].

The translation of a variable \( X \) of type \( t \), where \( t \) is either a Boolean value or a natural number, is given by

\[
\llbracket \text{var} X : t \rrbracket \overset{\text{def}}{=} \text{Loc}_X
\]

where

\[
\begin{align*}
\text{Loc}_X & \overset{\text{def}}{=} \text{w}_X(y).\text{Reg}_X(y) \\
\text{Reg}_X(y) & \overset{\text{def}}{=} \text{f}_X.\text{Reg}_X(y) + \text{w}_X(z).\text{Reg}_X(z)
\end{align*}
\]

\( \text{Loc}_X \) is the representation of a memory location for the variable \( X \). The value stored may be read along \( r_X \), and assigned along \( w_X \). The treatment given to variables of type \textit{reference} is slightly different. Two constant names are assumed,
\textit{NIL} and \textit{REF}, where the latter is used as an indication that the value stored is not \textit{NIL}. A match construct is thus necessary. We omit the details.

A sequence of variable declarations is translated as a composition of the agents representing each declared variable:

\[ [\text{var } X_1 : t_1, \ldots, X_n : t_n] \overset{\text{def}}{=} [\text{var } X_1] \cdots [\text{var } X_n] \]

An expression \( E \) is represented as an agent \([E](v)\), where \( v \) is the link along which the agent may communicate its value. We leave out the details.

An statement \( S \) is represented as an agent \([s](w)\) parametrised on a channel \( w \) representing the object where the statement occurs. Statements of type assignment \([X := E]\) are translated as the evaluation of \( E \) followed by the passing to \( \text{Reg}_X \) of a link to the agent denoting the value of \( E \). We omit details. Composition of statements \( P; Q \) is given by the derived operator \textit{before}

\[ P \text{ before } Q \overset{\text{def}}{=} (\text{done})(P \mid \text{done}(z)\, Q), \quad \text{where } z \notin \text{fn}(Q) \]

assuming that \( P \) is well-terminating, i.e. the last action of \( P \) is \textit{done}. But note that the agent \( P \text{ before } Q \) itself cannot execute \textit{done}, and is thus not well-terminating!

A method has shape

\begin{center}
\textbf{method } M(X,Y) \text{ is } S
\end{center}

where \( X \) is the single formal parameter supplied, and \( Y \) is a variable in which the value returned by the method invocation is stored. For each method name \( M \) we assume a constant name \( m \) of the \( \pi \)-calculus. The translation of a method is parametrised on the method name as is defined as follows:

\[ [\text{method } M(X; Y) \text{ is } S](m) \overset{\text{def}}{=} \overline{m}(z) \ast M(z) \]

where

\[ M(z) \overset{\text{def}}{=} (N)(\text{Loc}_X | z(w).z(x).\overline{w}x.([S](w) \text{ before } r_Y(u').\overline{x}u')) \]

with \( N = \{r_X, w_X, r_Y, w_Y\} \), the access sort of \( X \) and \( Y \). The expression \( \alpha \ast P \) denotes a replicator, i.e. \( !\alpha.P \). The agent

\[ z(w).z(x).\overline{w}x.([S](w) \text{ before } r_Y(u').\overline{x}u') \]

first obtains a link \( w \) to the object executing the method invocation (\texttt{self}), and a link \( x \) to the actual argument, which is stored in \( X \). Then the body of the method is executed, parametrised on \( w \), i.e \texttt{self}. Finally the result, assumed to have been stored in \( Y \), is returned along the link \( z \).

The expression \( E_1!M(E_2) \) signifies the sending of a message to the object obtained by the evaluation of the expression \( E_1 \), invoking the method \( M \) with the value of the expression \( E_2 \) as argument. The translation of this expression yields an agent that evaluates \( E_1 \) to \( z \), a link to the receiving object, and \( E_2 \) to \( x \), whereupon it executes \((u)\overline{w}u.\overline{m}.\overline{w}.P \), which means that a fresh link \( u \) to the object represented by \( w \) is sent, followed by the transmission the name of the method \( m \) and the value \( v \) along this \( u \). The \textit{continuation} \( P \) will roughly get the returned value along \( u \) and pass it on.

Corresponding to message sending there is a construct encoding message receipt, i.e. \texttt{answer} \((M_1, \ldots, M_k)(w)\), where \( w \) is again the link to the object executing the statement, i.e the object that receives the message. It is translated as
\[ \text{answer} \ (M_1, \ldots, M_n) (w) \overset{\text{def}}{=} w(u).u[m_1].\ldots.w[m_n].v.\forall v.\exists v'.\exists m[v'] [\text{done}] + \]
\[ \ldots + \]
\[ w(u).u[m].\ldots.w[m_n].v.\forall v.\exists v'.\exists m[v'] [\text{done}] \]

In this context, it is useful to look at the translation of the expression `self`:

\[ [[\text{self}(w, v)]] \overset{\text{def}}{=} \nu \text{REF} \ \! w \]

In contrast to value expressions, expressions of reference type are parametrised on two names. The reason is that it must keep track of the link \( w \) representing the pseudo-variable `self`. The result is the transmission of the link \( w \) along \( v \). This is all that is needed to represent self-referencing within an object.

Method declarations

\[ M\text{dec} : = M\text{dec}_1, \ldots, M\text{dec}_n \]

are translated by

\[ [[M\text{dec}_1, \ldots, M\text{dec}_n]] (m_1, \ldots, m_n) \overset{\text{def}}{=} [[M\text{dec}_1]] (m_1) \ldots \ | \ M\text{dec}_n (m_n) \]

and class declarations by

\[ [[\text{class } C \text{ is } V\text{dec}, M\text{dec} \text{ in } S] (c)] \overset{\text{def}}{=} (\nu c) (w) * [[V\text{dec}, M\text{dec} \text{ in } S]] (w) \]

The name \( c \) is a constant representing the class \( C \), while \( S \) signifies an statement. We have also

\[ [[V\text{dec}, M\text{dec} \text{ in } S]] \overset{\text{def}}{=} (N) [[V\text{dec}]] \ | \ M\text{dec} \ | \ [[S]] (w)) \]

where

\[ N = \{ r_x, w_x \mid X \text{ occurs in } V\text{dec} \} \cup \{ m \mid M \text{ occurs in } M\text{dec} \} \]

Each statement is encoded as an agent \([S](w)\), where \( w \) refers to the pseudo-variable `self`, i.e. the reference to the object of which \( S \) is a part. The task of the agent that encodes a class \( C \) is to provide on request, along the corresponding channel \( c \), an instance of the agent that encodes an object of class \( c \), i.e. \([V\text{dec} \text{ in } S]]\).

A program

\[ P\text{dec} \overset{\text{def}}{=} \text{program } P \text{ is } C\text{dec}_1, \ldots, C\text{dec}_n \]

is translated by

\[ [[P\text{dec}]] (w) \overset{\text{def}}{=} (c_1, \ldots, c_n) ([C\text{dec}_1] (c_1) \ldots \ | \ [C\text{dec}_n] (c_n) \ | \ c_1 (w).0) \]

The computation is initiated by the trigger \( c_1 (w).0 \).

As we may observe, for each method name there corresponds a constant name of the \( \pi \)-calculus. The agent that encodes a method requests the link to the object where the method is invoked. This link corresponds to the variable `self`. A private name is also passed for the reply. The correct method is invoked by matching the constant name of the method with the received name, whereupon the requested method is invoked along the channel denoting the method name. This scheme, however, does not work because there are no facilities for restricting the scope of the method name.
Thus, by interference, a method with the same name but belonging to another object might be erroneously invoked. In order to avoid this kind of interference, the names of the methods are restricted and made private for each object created. Unfortunately this scheme does not work either, because in this case only internal invocations of methods are possible. If an object invokes the method named $m$ in another object, the matching of the name $m$ to the internal method name in the receiving object will always yield false. This gives a good illustration of the difficulties deriving from the double nature of method names, which are public but whose significance is defined locally. This is basically the essence of late binding, which is a form of dynamic binding. The lack of primitives in the $\pi$-calculus for encoding dynamic binding complicates the solution to this problem.

In [180] another translation is given along the same lines of a version of $POOL$ [16], but now into a version of the polyadic $\pi$-calculus. A correspondence between the translation and an operational semantics of the translated language is also established. The problem concerning the scope of the names of the methods is solved here by the resorting to position, an approach that assumes a predefined protocol known the interacting parties.

This solution is cumbersome and cannot be extended to include inheritance. This procedure exemplifies once more the limitations of a calculus that lacks dynamic binding. In object-oriented languages it is the name of the method that constitutes the interface of an object, not the position that the method’s name arbitrarily occupies in a list of methods. In the presence of inheritance the number of methods might even be unknown to an invoking object at compilation time. Abstractly, an object responds to a message by invoking its version of the method specified in the message. Method invocation may thus be viewed as internal interaction along the channel representing the name of the method, whose scope is thus restricted within the object. This restriction is all that is needed in order to encode late binding.

Another work showing a $\pi$-calculus encoding of an object-oriented language is [130]. The target language is here a subset of Smalltalk-80 [56] that the author calls $Chat$. In this case the target language includes inheritance, return statements and block objects. By contrast, other important features in Smalltalk-80 such as the notion of a class as a kind of object, class variables, and runtime redefinition of methods are absent in $Chat$.

The problem with the scope of method names discussed above was solved here by letting those names be used only for matching purposes. Thus, these names are never used as channels, only as tokens for the methods. When an object receives a message with a method name, it proceeds to select, on the basis of this name, the corresponding method from an internal list of methods, which are identified by private names. The method is subsequently invoked along its internal name, i.e. a name under the restriction operator. There is thus for each object a mapping from the global to the internal names of the methods.

Inheritance is modelled by letting an object be represented as a composition of two agents, one representing the superclass and other the class itself. For this purpose each class is assigned not one but two names, one for the class itself and other for the class in the role of superclass. An object of class $ssc$ inheriting from a class $c$, with superclass name $sc$, simply creates along the channel $sc$ a superobject, i.e. a private object of class $c$. The difference between basic objects and superobjects is that the latter binds the pseudovariable $self$ to a channel representing the basic object, thus enforcing late binding. This channel is also used as a communication channel to the basic object.

Nevertheless, there are also problems here concerning once again the scoping of
names. In contrast to method names, instance variables have names that ought to be private to the object, and thus may not be referred from the outside. This is suitably done by using the restriction operator. In the solution presented here, variables are modelled as registers, exactly as in Walker’s version presented above. The access sorts of these variables are restricted within each object. This means that methods defined or redefined in a subclass may not refer to instance variables defined in the superclass. This example shows once again the subtle scoping issues that object-oriented languages pose. This is due to the phenomenon of encapsulation in object-based systems, which may be defined as a collection of concurrent objects, each endowed with a private environment, and which communicate with each other via message passing.

A different solution is presented in [158], which also uses the \( \pi \)-calculus to encode the dynamic behaviour of objects. In this model, the interface names of an agent representing an object are restricted, and communication between agents along these ports is not possible, as pointed out by the author. In order to allow this type of communication, the agents denoting objects are endowed with the capability of transmitting their local names, along a channel identifying the object, whenever requested by any party. Thus if we let \( \text{id} \) denote the identifier of an object, \( \bar{m} \) a vector of the method names, and \( \bar{v} \) the instance variables, the behaviour of an object \( OB \) may be recursively defined as

\[
OB \equiv (\bar{m}, \bar{v})(\ldots + \bar{id}(\bar{m}). OB(id, \bar{m}, \bar{v}))
\]

This solution is ultimately based on the position principle, since the agent requesting the local names on an object must know a priori the position of the method or methods that will be eventually selected, as well as the total number of methods defined for the corresponding class.

**Modeling Objects in PICT**

In [135] Pierce and Turner use a fragment of PICT to introduce an object-based programming style. PICT is a statically-typed programming language with a core calculus based on the asynchronous, choice-free \( \pi \)-calculus, extended with tuples and extensible records. A number of derived high-level forms are also defined by encodings into the core calculus.

The problems that derive from the absence of structuring primitives in process calculi are highlighted. In such calculi, concurrent systems are described as a “process soup”, “an unstructured collection of autonomous agents communicating in arbitrary patterns over channels.” However, programs usually contain more structure than what this picture shows. A more suitable picture would be that concurrent systems consist of collections of groups of processes that cooperate by sending messages and maintaining locally a number of invariants, thus presenting to the rest of the environment an interface that is described as “an abstraction of a shared internal state.” Each group of processes may be viewed as consisting of objects of a certain type or class that determine the kind of interface offered to the rest of the world. However, due to the absence of more powerful structuring primitives in process calculi, as mentioned above, “these groups of processes exist nowhere but in the programmer’s mind.” To make them real, the authors propose the introduction of a series of structuring primitives intended to enable a program “to refer explicitly to groups of processes,” which are thus given “the status of rudimentary objects.”

Objects are modelled as sets of persistent processes encoding variables and methods. The interface of an object is a record containing the channels that give access to the
exported features. The authors justify the introduction of records by the argument that “referring to a group of processes consists exactly in referring to a collection of channels where these processes are listening,” i.e. the object’s interface. Records are suitable for this purpose because they allow “members of a group of channels to be selected by name” and provide a “well-defined interface through which a set of related services may be accessed.” Records provide also higher-order features because “the complete interface to an object may be manipulated as a single value.”

The authors consider that records and tuples can be encoded in pure π-calculus, but that the encoding would not preserve enough structure to support the typing rules they expect for the language. Reasons of efficiency are also named, e.g. that in the presence of single-field extension of records the relative position of the fields might not be known at compile-time.

The syntax of core language is quite simple. The entities, or values, that can be communicated on channels are the following:

\[
\begin{align*}
\text{Val} &= \text{Id} & \text{variable} \\
&= [\text{Val}, \ldots, \text{Val}] & \text{tuple} \\
&= \text{record end} & \text{empty record} \\
&= \text{Val with Id=Val end} & \text{record extension} \\
\text{Pat} &= \text{Id} & \text{variable pattern} \\
&= [\text{Pat}, \ldots, \text{Pat}] & \text{tuple pattern} \\
&= \text{record Id=Pat, \ldots, Id=Pat end} & \text{record pattern} \\
&= - & \text{wildcard pattern} \\
\text{Abs} &= \text{Pat} > \text{Proc} & \text{abstraction} \\
\text{Dec} &= \text{new Id} & \text{channel creation} \\
\text{Proc} &= \text{Val}?\text{Abs} & \text{input prefix} \\
&= \text{Val}?*\text{Abs} & \text{replicated input} \\
&= \text{Val}!\text{Val} & \text{output atom} \\
&= \text{Proc}|\text{Proc} & \text{parallel composition} \\
&= \text{let Dec in Proc end} & \text{declaration}
\end{align*}
\]

Val are the entities or values, that can be communicated on channels. Pat are the patterns that may occur in input prefixes; wildcard patterns matches any value. Abs are abstractions, a process prefixed by a pattern. Dec is a declaration providing a fresh channel Id, and Proc are the processes. The scope of a declaration let Dec in Proc end denotes the process Proc, but may be extruded as in the π-calculus.

The multi-field record

\[
\text{record} l_1 = v_1, \ldots, l_n = v_n \text{ end}
\]

is defined as syntactic sugar for

\[
\text{record end with } l_1 = v_1 \text{ end … with } l_n = v_n \text{ end}
\]

The record extension v with \(1=v'\) creates a copy of v where the field labelled 1 has value \(v'\); if the field already exists in v its value is simply modified.

The semantics is defined with the help of a structural congruence stating that parallel composition is commutative and associative, and that fresh channels may be extruded if conveniently renamed. The rules of reduction are as expected: channel-based communication rules, rules allowing reduction within declarations, a parallel
reduction rule and a structural reduction rule. The main difference with the standard \( \pi \)-calculus is that the communication rule is allowed only if the input pattern matches the output value, in which case a resulting substitution is applied to the continuation of the input-prefix fixed process taking part in the communication. The labels of a record pattern must match one the the labels of a record value, but a label in the the record value that does not occur in the record pattern is simply ignored. Values and patterns corresponding to identical labels must match too. For details see [135].

As an example of how an object is implemented in PICT, the following program fragment is presented:

```plaintext
ref?[init, res] >
  let
    new contents, s, g
  in
    contents!init
    | res!record set=s, get=g end
    | (s?[v, c] > contents?v > contents!v | c![])
    | (g?[r] > contents?x > contents!x | r!x)
  end
```

This process implements an integer reference cell class. It continuously listens on the channel \( \text{ref} \) for a pair of parameters representing an initial value \( \text{init} \) to be stored in the cell and a reply channel \( \text{res} \) for communicating the result of the call, a record denoting the object created by the call. Each new object consists of one instance variable, \( \text{contents} \), and two methods \( s \) and \( g \). Each one of these is represented by a fresh name, but the name of the methods are extruded. Note that the labels of the resulting record, \( \text{get} \) and \( \text{set} \), must be public and previously known to the agent invoking the creation of a new reference cell object.

This scheme illustrates once again the limitations of the \( \pi \)-calculus that derives from the lack of structuring and localisation primitives. In order to differentiate between distinct reference cells, each object must be endowed with its own version of the channels representing the methods \( \text{get} \) and \( \text{set} \), apart from the instance variable \( \text{contents} \). However, in the absence of the matching operator, the method channels must be extruded, thus compromising their integrity. As an example, a process that receives a version of the method channel \( s \) might use it as an input channel, and may thus interfere with invocations to the reference cell object made by other agents sharing the cell. Even worse, this could interfere with invocations to methods made by other methods in the same object, including recursive ones. A sensible requirement is that at least this type of invocation should be safe. Also, the identity of the object is not clear, since the resulting record, which is meant as a reference to the object, can be decomposed by matching and the extruded channels \( s \) and \( g \) may be further publicised separately. An agent could thus in principle own the capability to read or write into the cell, but not both.

Pierce and Turner’s model captures essential aspects of concurrent object such as encapsulation, identity, persistence, instantiation and synchronisation. Nevertheless, other important aspects are not covered, e.g. self-reference, late binding, inheritance and reflection. With the purpose of incorporating these basic features, an extension of the model is presented in [97, 154, 153]. This is done by introducing metaobjects as first-class entities. The treatment given to method names are nevertheless the same, and the same remarks we made above in relation to this issue apply also here.
Modeling Objects in TyCO

Tyco [174] is an implicitly typed calculus developed with the aim of modelling concurrent objects that communicate via asynchronous message passing. TyCO was inspired by the π-calculus, Honda’s ν calculus [70], and Hewitt’s actor model [66]. The notions of object, asynchronous messages and concurrency are primitive in TyCO. The basic approach of TyCO is to “build high-level constructors present in functional and concurrent object-oriented languages” given a name-passing calculus [173]. By contrast, the work of Walker [179], discussed in Section 6.3.3, starts from an existing language and produces a systematic translation into the π-calculus.

The terms of the calculus are built from names and few constructors. Messages have shape $a \cdot l(\bar{v})$, where $a$ is the name of an object at which the message is directed, $l$ is the name of the method selected, and $\bar{v}$ a sequence of names denoting the arguments to the method denoted by $l$. An object located at $a$ is represented by a term of form

$$a : [l_1(\bar{x}_1).P_1 & \ldots & L_n(\bar{x}_n).P_n]$$

where for each $i$ such that $1 \leq i \leq n$, $l_i$ denotes a method, $\bar{x}_i$ a sequence representing the formal parameters of the corresponding method $l_i$, and $P_i$ the body of the method. The other primitives are a composition operator, scope restriction, and replication. Objects are supposed to have a single identifier. The syntax assumes the existence of a both set $N$ of names and a set $L$ of labels. A typical interaction involving the agents $a \cdot l_k(\bar{v})$, where $k \in \{1, \ldots, n\}$, and $a : [l_1(\bar{x}_1).P_1 & \ldots & L_n(\bar{x}_n).P_n]$, yields the agent $P_k(\bar{v}/\bar{x}_k)$.

A translation of TyCO to the polyadic π-calculus is given. The idea is to give each identifier and each label a distinct name. For instance, an object

$$a : [l(\bar{x}).P & m(\tilde{y}).Q]$$

is translated into

$$a_l(\bar{x}).P + a_m(\tilde{y}).Q$$

whereas a message $a < l(\bar{v})$ becomes $a_l.\bar{x}.0$.

Note that this translation assumes the existence of specialised names. An alternative offered is to have a two-step protocol, as in [158], presented in Section 6.3.3. In this case, a message first asks the object method names, which are assumed to be restricted, and then selects one of them. As pointed out by the authors, “the drawback of this encoding is that it assumes we know all the methods in the target object, and therefore it is not context independent” [174]. What seems clear is that there is always a price to pay in avoiding the use of specialised names and the information they bear, which usually must be rendered in terms of complicated protocols that moreover impairs flexibility and extensionality. As noted by the authors, Walker’s translation [179], discussed in Section 6.3.3, assigns “each method in an object a different name, and multiplex all these names into a single name,” thus making “invocation of a method a three-way protocol” [173].

Abacus, OL1 and the Object Calculus (OC)

In [129] and [128], an executable notation called Abacus is presented. Abacus is based on CCS [109] and CSP [69] and is designed for specifying and prototyping object-based concurrent languages. The intention is to use a process calculus as a semantic foundation for concurrent object-oriented languages. An object is viewed
as a process consisting of patterns of agents that obey a protocol established by a programming language whose specification is given as a “mapping from syntactic patterns representing the language constructs to the behavioural patterns that they stand for.” [127]

Agents in Abacus are described in terms of possible communications with other agents. The notions of class, inheritance, and genericity are modelled as patterns of behaviour. The authors claim that objects are naturally modeled as communicating agents irrespective of the particular object model or whether objects are active or passive. An object is an agent with a unique identity that communicates with other objects via call and reply messages. Agents are described as “entities that may change state when it communicates with another agent.” Communication in Abacus is synchronous.

The notation includes prefix output and input, the choice and composition operators, and the null process. Two categories of names are present: event names and agent names. Synchronisation happens along event names. Messages are modelled as compound events, i.e., list or tuples. Additional operators are included for modelling encapsulation. These include restriction, relabelling, label prefixing and filtering. Restriction and relabelling are used for hiding or relabelling selected offers, whereas filtering is used to “define scopes beyond which only ‘prefixed offers’ are visible” and “to hide all but selected offers to communicate with the outside world” [129]. This is the only example we know of the use of a filtering operator in a process algebra notation. An object’s instance variables are protected from being accessed by other objects with the help of the restriction operator.

By way of conclusion, it is suggested that most of the difficulties associated with concurrency and inheritance result from “a lack of good methods for encapsulating objects,” a fact that we believe is confirmed by the many errors and cumbersome solutions that have been proposed as solutions to these problems.

A similar early notation, OLI, intended for exploring object-based concepts in a process calculus framework based on CCS and CSP was presented in [119]. As in Abacus, objects are endowed with unique object identifiers and communication between named objects takes place along compound event structures.

In [127] a new object calculus “borrowing heavily from process calculi,” the Object Calculus, briefly OC, is put forward by Nierstrasz. OC is described as an evolution of Abacus, bearing “the same relationship to Abacus that the π-calculus does to CCS,” according to the author. It is described as a unification of of the π and λ-calculi. It extends the π-calculus with tuple-based communication, functional application and communication of expressions. The goal of OC is to integrate the concept of agent in process calculi with the concept of function in the λ-calculus. Active objects are viewed as functions or agents with state. However, pattern communication restricts extensibility and reusability even more than polyadic communication, and requires the specification of detailed protocols that must be followed to by all the agents in the system.

**Modeling Components in πŁ**

Components are often defined as black-box entities with well-defined interfaces encapsulating services or “plugs”. Composition of components requires interfaces that are plug-compatible as well as the presence of standardised interaction protocols. A suitable mathematical foundation for components representing open systems should include concurrency and communication as primitives. To this end, an extension of the π-calculus, called πŁ, was proposed in [96]. The starting point for this calculus
is the observation that the tuple-based communication of e.g. the \( \pi \)-calculus and the Object Calculus \[127\] restricts code extensibility and reuse. Inspired by the \( \lambda N \) calculus of Dami \[42, 40\], which was discussed in Section 6.2.5, communication in \( \pi L \) is defined on the basis of \( form \), a kind of extensible record. In \( \lambda N \) parameters are identified by names rather than positions. Analogously, parameters in \( \pi L \) are identified by names, and communication of tuples is substituted by communication of labelled parameters.

As an illustration we show here an example extracted from \[10\]. A process in the polyadic \( \pi \)-calculus that provides, along channel \( w \), a service parametrised by \( a \) and \( b \), and that behaves as a wrapper for another process that provides a service along channel \( f \), may be defined by

\[
!w(a, b, r). (\nu r')(\overline{f}(a, b, r') \mid r'(x, y)). \overline{\rho}(x, y)
\]

Channels \( r \) and \( r' \) above are reply channels. This example may be encoded in \( \pi L \) as

\[
!w(X). (\nu r')(\overline{f}(X{\text{reply}} = r')) \mid r'(Y). \overline{X{\text{reply}}}(Y)
\]

The wrapper in this expression expects a single form \( X \), which is passed by \( f \) with the reply channel \( r \) overridden by \( r' \). The reply \( Y \) is finally transmitted along the channel \( X{\text{reply}} \). The wrapper in this case is generic and assumes only that the message contains a reply channel.

What distinguishes the \( \pi L \) calculus from \( \lambda N \) is that the former is based on the \( \pi \)-calculus, wherein the notion of names is basic. However, the identification of names with channels is weakened in the \( \pi \)-calculus by the presence of the matching operator, which treats channels as pure names. In the same way, the labels appearing in a form in the \( \pi L \) calculus are also pure names, their only purpose being to indicate the location the parameters in a form. Extraction of parameters in forms, as in records, is done by matching labels. It is assumed that the name of parameters are global, and cannot be \( \alpha \)-converted since they constitute the basic means of communication among interacting agents. Names in the \( \pi \)-calculus can also play the same role, as the example below shows.

Boolean values and negation may be encoded in the polyadic \( \pi \)-calculus as follows:

\[
\begin{align*}
\text{True}(r) &= (\nu b)(\overline{\rho}(b) \mid b(t, f). \overline{t}) \\
\text{False}(r) &= (\nu b)(\overline{\rho}(b) \mid b(t, f). \overline{t}) \\
\text{Not}(b, r) &= (\nu t, f)(b(c). \overline{\rho}(t, f) \mid t. \text{False}(r) \mid f. \text{True}(r))
\end{align*}
\]

It is easy to verify that

\[
(\nu a)(\text{Not}(a, r) \mid \text{True}(a)) \simeq \alpha \text{false}(r)
\]

where \( \simeq \alpha \) denotes asynchronous weak equivalence \[14\]. For details see \[96\]. Now, if we want to extend the system to a three-value logic, everything has to be recompiled as follows:

\[
\begin{align*}
\text{True}_U(r) &= (\nu b)(\overline{\rho}(b) \mid b(t, f, u). \overline{t}) \\
\text{False}_U(r) &= (\nu b)(\overline{\rho}(b) \mid b(t, f, u). \overline{t}) \\
\text{Unknown}_U(r) &= (\nu b)(\overline{\rho}(b) \mid b(t, f, u). \overline{t}) \\
\text{Not}_U(b, r) &= (\nu t, f, u)(b(c). \overline{\rho}(t, f, u) \mid t. \text{False}_U(r) \mid f. \text{True}_U(r) \mid u. \text{Unknown}_U(r))
\end{align*}
\]
As shown in [14], this encoding is not compatible with the former one. For instance

\[(\nu a)((\text{Not}(a, r) \mid \text{True}(a)) \neq \text{false}(r))\]

since communication may occur only if the arities of the input and the output actions agree.

The solution is to introduce a naming scheme for parameters, or forms, which are immutable mapping from labels to names. Boolean values can thus be encoded as follows:

\[
\begin{align*}
\text{True}(r) &= (\nu b)(r(\langle \text{val} = b \rangle) \parallel b(X).X_{\text{true}}) \\
\text{False}(r) &= (\nu b)(r(\langle \text{val} = b \rangle) \parallel b(X).X_{\text{false}}) \\
\text{Not}(c, r) &= c(Y).(\langle \nu \text{val} = b \rangle) \parallel b(X).Y_{\text{val}}(X(\text{true} = X_{\text{false}})(\text{false} = X_{\text{true}})))
\end{align*}
\]

As pointed out in [96], in $\pi L$ “we need to obey some naming discipline. Whenever two $\pi L$-agents are willing to communicate they must also agree on the set of labels they want to use.” In the case of Boolean values described above, the naming discipline involves the use of $\text{true}$ and $\text{false}$ as pure names. Now nothing prevents us from introducing this naming discipline in the $\pi$-calculus itself by decreeing that $\text{true}$ and $\text{false}$ are to be used as names that denote the Boolean variables. In this case, $\text{True}(r)$ may be defined simply as $r(\text{true})$, and $\text{False}(r)$ as $r(\text{false})$. Negation may be defined by

\[
\text{Not}(c, r) = c(\text{val}).([\text{val} = \text{true}]r(\text{false}) + [\text{val} = \text{false}]r(\text{true}))
\]

This illustrates how name specialisation, which determines a kind of protocol, enhances flexibility in communicating systems. The protocols determined by name specialisation may easily be extended by the addition of new specialised names.

Based on the semantic foundation offered by the $\pi L$ calculus and the communication of forms, a language called Piccola was outlined in [10]. Piccola encodes components and compositional abstraction as communicating concurrent agents, and interfaces of components as forms, which unifies several concepts in the language and enhances expressiveness [9]. Piccola includes constructs defining functions, infix operators, and dynamic contexts “to encapsulate required services” [10]. A component offers a set of named services along its interface. Program fragments are called scripts, and agents evaluate a script to a form with a set of bindings that is made public. Contexts are encoded as forms, by which two kinds of namespace are defined: the root namespace representing the lexical scope, and dynamic namespaces representing environments passed at invocation time. Services are forms that may carry additional information and allow the treatment of functions as first-class values. Arguments for services are also represented by forms that provide keyword-based argument passing.

The Piccola language does not explicitly model localities, and components cannot be composed dynamically. It is implemented by translation into the $\pi L$ calculus. According to the authors, “the notion of forms has evolved on top of the $\pi$-calculus.” Although it is claimed that “forms can be explained as an abstract datatype, independent of names” [9] this work seems rather to strengthen the thesis that the notions of naming and environments are essential in open systems and in notations intended to give a formal description of this kind of system.

192
6.4 Conclusions

In this chapter we have seen that names and dynamic binding of names are ubiquitous in systems that involve interaction and communication, e.g., object-based systems, incremental programming environments, modular and distributed systems. We have argued that the functional paradigm has a bias against such notions as extensibility, sharing, state, persistency, and behaviour. We have seen that the notion of value, central in the functional paradigm, is not strictly necessary to describe computation, and can be traded for a more intensional and behavioural notion such as continuation-passing. We have observed also that there is often a trade-off between self-referential constructs and dynamic table lookups, and have drawn a line between self-reference and recursion. Since interactional or reactive computation is intrinsically intensional, it is best captured, in a intuitive sense, by an operational semantics.
Chapter 7

Conclusions

The major objective of this thesis has been to establish the significance of naming and dynamic binding of names in communicating and mobile systems, and to give a formalisation of these notions. We have shown that these concepts, whose meaning and importance have been assessed only fragmentarily in previous work, are central for those systems. We have argued against the denotational and functional approaches towards this subject, and in favour of the so-called object paradigm. We have extended the $\pi$-calculus with operators for dynamic binding, and developed a notation based on the reflexive Abstract Chemical Machine, extended with primitives encoding the notion of dynamic binding, in which features associated with object-based systems have been modelled.

The position that higher-order features in concurrency can be reduced to first-order ones by passing names as triggers of processes has been reexamined, and found to be inadequate in the presence of dynamic binding mechanisms. Since mobile code, a higher-order feature, is a reality in mobile systems, and since dynamic binding of names occurring in mobile code enhances both flexibility and security, we conclude that formalisms that lack primitives for encoding dynamic binding as well as higher-order features are insufficient as computational models for mobile systems.

We have demonstrated that dynamic binding may be encoded in process algebraic formalisms by primitives such as the blocking and the filter operators, which delimit the range of names that occur within their scope. Moreover, we have shown how these operators may be used to capture the idea of locality in mobile systems, and to reflect the structure induced by the linkage among system components.

The notion of blocking has been refined by the introduction of polarised blocking, which has been shown to be useful for encoding runtime control of resource access and security-related notions such as capability, protection domains and access rights. We have argued that the use of static channel types and static process types for modelling access right constraints has many limitations, and have shown how polarised blocking and filtering may be used to encode dynamic checking of access rights.

We have developed a calculus, called CRCHAM, constructed basically by adding polarised blocking to a notation similar to the reflexive Chemical Abstract Machine. We have shown that polarised blocking in CRCHAM adequately encodes the notion of environment. The combination of join pattern mechanisms to encode state and of polarised blocking to encode environments has been shown to be a powerful one for encoding object-based mobile systems. A higher-order version of CRCHAM has been used to model the concept of reflection and metalevel objects in the context of a strict hierarchical class structure similar to the one provided in Smalltalk.
Finally, in keeping with our conviction that computer science is basically an empirical enquiry, we decided to support our claim about naming and dynamic binding on a detailed examination of how these notions manifest themselves in existing systems and formalisms. Our conclusion is that the absence of dynamic binding mechanisms often results in awkward encodings of objects in mobile systems, and that the functional and denotational approaches cannot convincingly capture dynamic aspects of open systems such as extensibility and persistence. The picture that steadily grew out of this examination strengthened our conviction that a semantics by translation into an executable notation, as provided by process algebraic formalism, yields a more suitable framework for giving a meaning to mobile system constructs.

In this study we have concentrated mainly on the expressiveness of our formal notations, e.g. the CR-CHAM, to encode dynamic binding in calculi for mobile systems. Further investigation would be necessary in order to develop a semantics that can help us to reason about such systems.

Another line of investigation that naturally follows from our observations about names and dynamic binding in mobile systems, and computation in general, concerns the question as to how these notions can be given a solid formal foundation. The concept of matching is crucial here. In synchronous calculi, such as the π-calculus, matching appears implicitly in the way complementary channels may engage in a communicating event, and explicitly in the matching operator. In an asynchronous setting without handshaking the association of a name to a channel is weaker, and negative prefixes are absent. Here, a negative action may be best regarded as a message. In other calculi, e.g. Linda, the concept of channel is also absent, and names are used in matching procedures involving tuples, i.e. name structures. These two distinct approaches towards communication and concurrency appears to be rooted in the way names are interpreted by each calculi. It is our belief that the association of names with channels should be viewed only as a particular case of a more general notion of name.

The notion of dynamic binding in computer languages is closely related to the notion of context in artificial intelligence [102] and in natural language [61]. Ordinary speech is highly contextualized [166]. On the borderline between linguistics and computer science we could mention the works by Janssen [73, 74]. We believe that an investigation of the works put forth in these fields might fertilize the study of communication and interaction in computer science.

What we are proposing is ultimately an interpretation of the concept of name as a distillation of the notion of symbol in physical-symbol systems, as defined by Newell and Simon [125]. A physical-symbol system consists of a set of entities, the symbols, that may be built into structures called expressions. Expressions are composed of instances of symbols or tokens. At any instant of time the system contains a collection of processes that operate on expressions to produce other expressions. Symbols may be used to designate any expressions, and the designation is not prescribed a priori. What we propose is a refinement of this picture by decomposing the total system into a hierarchy of environments binding names to structures, and within which the processes would operate. We believe that a theory of mobile computation can be defined in terms of symbol manipulation within a symbol system.

We use the term computation here in a broad sense, covering not only sequential computation but also interactional computation. The development of such a theory of computation is a much more modest endeavour than what has been otherwise the central focus of the debate over the so-called physical symbol system hypothesis, i.e. whether general intelligent action may be defined in terms of symbol manip-
ulation in a physical symbol system. We do not go as far and want to raise only the question as to whether general computational behaviour, as exhibited by networks of computers and computing agents that communicate with each other, may be defined in terms of symbol manipulation and formalised by a name calculus. Although this question about computation seems to be less pretentious than the one about intelligence, we should not forget the fact that theories about the nature of interactional computation seems to enjoy as little consensus among scholars as those concerning the nature of intelligence.
Bibliography


203


Appendix A

Appendix

A.1 Appendix to Section 3.2

Lemma 2.1 If $P \xrightarrow{a} P'$ then

(i) $\text{fn}(a) \subseteq \text{fn}(P)$, and

(ii) $\text{fn}(P') \subseteq \text{fn}(P) \cup \text{bn}(a)$.

Proof: By induction on depth of inference. As in Lemma 1 in [112], extended with a new case, BLOCK. Assume $P \equiv Q \setminus z \xrightarrow{a} Q' \setminus z \equiv P'$. Then, by BLOCK, $z \not\in n(a)$, and by a shorter inference $Q \xrightarrow{a} Q'$.

(i) holds since by induction $\text{fn}(a) \subseteq \text{fn}(Q) \subseteq \text{fn}(Q \setminus z) = \text{fn}(P)$.

(ii) holds since by induction $\text{fn}(Q') \subseteq \text{fn}(Q) \cup \text{bn}(a)$, hence $\text{fn}(P') = \text{fn}(Q' \setminus z) = \text{fn}(Q') \cup \{z\} \subseteq (\text{fn}(Q) \cup \text{bn}(a)) \cup \{z\} = \text{fn}(Q \setminus z) \cup \text{bn}(a) = \text{fn}(P) \cup \text{bn}(a)$.

Lemma 2.2 Suppose that $P \xrightarrow{a(y)} P'$, where $a = x$ or $a = \overline{x}$ and that $z \not\in n(P)$. Then equally for some $P'' \equiv_a P' \{z/y\}$, $P' \xrightarrow{a(z)} P''$.

Proof: Induction on depth of inference, as in Lemma 2 in [112], extended with the case BLOCK: let $P \equiv Q \setminus w$. Then, by BLOCK, $P' \equiv Q' \setminus w$ for some agent $Q'$ such that $Q \xrightarrow{a(w)} Q'$ by a shorter inference, $w \neq x$ and $w \neq y$ since $w \not\in s(a(y)) \cup \text{bn}(a(y)) = \{a, y\}$. By assumption $z \not\in n(P) = n(Q \setminus w) = n(Q, w)$.

Hence $z \not\in n(Q)$. By induction, for some agent $Q'' \equiv_a Q' \{z/y\}$, $Q \xrightarrow{a(z)} Q''$. Note that by definition of $a$-equivalence we get $Q'' \setminus w \equiv_a Q' \{z/y\} \setminus w$. Also $z \neq w$, thus $w \not\in s(a(z)) \cup \text{bn}(a(z)) = \{a, z\}$. Then by BLOCK $P \equiv Q \setminus w \xrightarrow{a(z)} Q'' \setminus w \equiv_a Q' \{z/y\} \setminus w \equiv (Q' \setminus w) \{z/y\}$ (since $w \neq z$ and $w \neq y$). Now, let $P'' \equiv Q'' \setminus w$ and we are done.

Lemma 2.3: If $P \xrightarrow{a} P'$, $\text{bn}(a) \not\in \sigma[\text{fn}(P')]$ and $\sigma[\text{fn}(P)]$ is injective, then equally for some $P'' \equiv_a P', \sigma[\text{fn}(P)]$ is injective.

Proof: Proof by induction on depth of inference.

In Lemma 3 in [112] there is no requirement that $\sigma[\text{fn}(P)]$ be injective. The requirement is necessary here, since otherwise the lemma is not true. For instance, consider $(\exists x: 0) b$ and $\sigma = \{b/a\}$. Nevertheless, it is enough to consider this restricted form of the lemma. Moreover, in Lemma 3 in [112] there is no requirement that $\text{bn}(a) \not\in \sigma[\text{fn}(P')]$. Instead, it is erroneously required that
Lemma 2.4 If $P\{w/z\} \xrightarrow{\alpha} P'$, where $w \notin \text{fn}(P)$ and $\text{bn}(\alpha) \cap \text{fn}(P, w) = \emptyset$, then equally for some $Q$ and $\beta$ with $Q\{w/z\} \equiv \alpha P'$ and $\beta \sigma = \alpha$, $P \xrightarrow{\beta} Q$.

Proof: By induction on depth of inference. For BLOCK we have $P \equiv P'' \backslash x$ for some $P''$, $(P'' \backslash x)\sigma \equiv P'' \backslash x \sigma \xrightarrow{\alpha} P'' \equiv Q' \backslash x \sigma$ for some $Q'$, $x \sigma \notin \text{fn}(\alpha) \cup \text{bn}(\alpha)$, and by BLOCK $P'' \sigma \xrightarrow{\alpha} Q'$ by a shorter inference. Since by assumption $w \notin \text{fn}(P) = \text{fn}(P'' \backslash x)$, then $w \notin \text{fn}(P''')$, and since $\text{bn}(\alpha) \cap \text{fn}(P, w) = \text{bn}(\alpha) \cap \text{fn}(P'' \backslash x, w) = \emptyset$, and $\text{fn}(P'' \backslash x, w) = \text{fn}(P', x, w)$, then $\text{bn}(\alpha) \cap \text{fn}(P'', w) = \emptyset$. Hence, by induction, there is a $Q'$ and a $\beta$ such that $Q' \sigma \equiv \alpha Q$, $\beta \sigma = \alpha$ and $P'' \xrightarrow{\beta} Q'$. Now, $x \notin \text{fn}(\beta)$ since $x \sigma \notin \text{fn}(\alpha) = \text{fn}(\beta \sigma)$. Moreover, $x \notin \text{bn}(\alpha) = \text{bn}(\beta)$ since $\emptyset = \text{bn}(\alpha) \cap \text{fn}(P'' \backslash x, w) = \text{bn}(\beta) \cap \text{fn}(P'', x, w)$. Hence, $x \notin \text{fn}(\beta) \cup \text{bn}(\beta)$. Then, by BLOCK, $P \equiv P'' \backslash x \xrightarrow{\beta} Q' \backslash x$ and $(Q'' \backslash x)\sigma \equiv Q'' \sigma \backslash x \sigma \equiv Q' \backslash x \sigma \equiv P'$. Now let $Q$ be $Q' \backslash x$ and the proof is complete.

Lemma 2.5: Suppose that $P \equiv \alpha Q$.

(a) If $\alpha$ is a free action and $P \xrightarrow{\alpha} P'$ then equally for some $Q'$ with $P' \equiv \alpha Q'$,

$Q \xrightarrow{\alpha} Q'$.

(b) If $P \xrightarrow{\alpha(y)} P'$, where $a = x$ or $a = x$, and $z \notin \text{n}(Q)$ then equally for some $Q'$

with $P\{z/y\} \equiv \alpha Q'$, $Q \xrightarrow{\alpha(z)} Q'$.

Proof: In the corresponding lemma in [112], Lemma 3, the version of Lemma 2.3 presented there was applied to an agent $P\sigma$ where $\sigma = \{v/w\}$ and $v \notin \text{fn}(P_1)$. But in this case $\sigma \backslash \text{fn}(P_1)$ is injective, and thus the same proof goes through by applying our version of this lemma, extended with the case for BLOCK.

For BLOCK we have $P \equiv P'' \backslash w$ and $Q \equiv Q'' \backslash w$ for some agents $P''$ and $Q''$ such that $P'' \equiv Q''$ since $P'' \backslash w \equiv Q'' \backslash w$ by assumption. Then $P \equiv P'' \xrightarrow{\alpha(w)} P' \equiv P'' \backslash w$ for some $P''$, $w \notin \text{n}(\alpha)$. Then, by BLOCK, $P'' \xrightarrow{\alpha} P''$ by a shorter inference.

(a) Assume $\alpha$ is a free action. Then, by induction there is an agent $Q'''$ such that $Q''' \equiv \alpha P''$ and $Q'' \xrightarrow{\alpha} Q'''$. Hence by BLOCK, $Q \equiv Q'' \xrightarrow{\alpha} Q''' \backslash w \equiv \alpha P'' \backslash w \equiv P'$, and we define $Q'$ as $Q''' \backslash w$.

(b) By BLOCK $w \neq x$ and $w \neq y$. By assumption, $z \notin \text{n}(Q') = \text{n}(Q'' \backslash w)$. Thus $w \neq z$ and $z \notin \text{n}(Q')$. Hence, by induction, there is a $Q''$ such that $P'' \{z/y\} \equiv \alpha Q''$ and $Q'' \xrightarrow{\alpha(z)} Q'''$. Since $w \neq z$ then $w \notin \text{n}(a(z)) \cup \text{bn}(a(z))$.

Hence, by BLOCK $Q'' \backslash w \xrightarrow{\alpha(z)} Q''' \backslash w \equiv \alpha P'' \{z/y\} \wedge (P'' \backslash w)\{z/y\}$ (since $w \neq y$) $\equiv P'\{z/y\}$. Now let $Q' \equiv Q''' \backslash w$ and the proof is complete.

Theorem 2.1 $\equiv \alpha$ is a strong bisimulation.

Proof: Proof as in [112], using Lemma 2.5 (Lemma 5 there).

Lemma 2.6: If $P \simeq Q$ and $w \notin \text{fn}(P, Q)$, then $P\{w/z\} \simeq Q\{w/z\}$.

Proof: Precisely as in [112], since Lemma 3 there is applied only to agents of form $P\sigma$ where $\sigma \backslash \text{fn}(P)$ is injective.

Lemma 2.7: If $S$ is a strong bisimulation up to restriction then $S \subseteq \simeq$.  

211
Theorem 2.2

(a) \(\sim\) is an equivalence relation.

(b) If \(P \sim Q\) then

\[
\begin{align*}
\alpha.P & \sim \alpha.Q \quad (\alpha \text{ a free action}) \\
P + Q & \sim Q + R \\
[x = y]P & \sim [x = y]Q \\
P | R & \sim Q | R \\
(w)P & \sim (w)Q \\
P \setminus z & \sim Q \setminus z
\end{align*}
\]

Proof: Precisely as in [112].

Lemma 2.8: If every agent identifier is weakly guardedly defined then, for any agent \(P\), there is a head normal form \(H\) such that \(\text{SGE} \vdash P = H\).

Proof: As in [112], with the help of H1-H3.

Theorem 2.5 (Completeness for finite agents): For all finite agents \(P\) and \(Q\), if \(P \sim Q\) then \(\text{SGE} \vdash P = Q\)

Proof: Precisely as the proof for Theorem 20 in [112].

Below, in view of Theorem 2.1 we assume all identities are up to \(\alpha\)-equivalence. We assume also that all substitutions \(\sigma\) are the identity except on a finite set of names, and that there are always fresh names available.

Lemma 2.9: For all substitutions \(\sigma\) and all processes \(P\), \(\tau(P)\sigma = \tau(P\sigma)\), and if \(\sigma(w) = w\), then \(\tau_{wz}(P)\sigma = \tau_{wz}(P\sigma)\).

Proof: by structural induction on \(P\), assuming all bound variables have been conveniently renamed.

\[\begin{align*}
\blacktriangle P & \equiv 0. \text{ Then } \tau(P)\sigma \equiv \tau(0)\sigma = 0\sigma = 0 = \tau(0) = \tau(0\sigma) \equiv \tau(P\sigma). \\
\blacktriangle P & \equiv \alpha.Q. \text{ Then } \tau(P)\sigma \equiv \tau(\alpha.Q)\sigma = (\alpha.\tau(Q))\sigma = \alpha.\tau(Q)\sigma = \alpha.\tau([\alpha.Q]\sigma) = \tau([\alpha.Q]\sigma) \equiv \tau(P\sigma). \\
\blacktriangle P & \equiv (x)Q. \text{ Then } \tau(P)\sigma \equiv \tau((x)Q)\sigma = ((x)\tau(Q))\sigma = (x)\tau(Q)\sigma = (x)(\tau(Q)\sigma) \text{ (by induction)} = \tau(\{(x)Q\}) = \tau((\{(x)Q\}) \equiv \tau(P\sigma). \\
\blacktriangle P & \equiv [x = y]Q. \text{ Then } \tau(P)\sigma \equiv \tau([x = y]Q)\sigma = ([x = y]\tau(Q))\sigma = [x = y]\tau(Q)\sigma = [x\sigma = y\sigma]\tau(Q)\sigma \text{ (by induction)} = \tau([x\sigma = y\sigma]Q)\sigma = \tau([x = y]\tau(Q)\sigma) = \tau([\{x = y]Q\sigma) \equiv \tau(P\sigma). \\
\blacktriangle P & \equiv (!Q). \text{ Then } \tau(P)\sigma \equiv \tau(!Q)\sigma = (!\tau(Q))\sigma = (!\tau(Q)\sigma) = !\tau(Q\sigma) \text{ (by induction)} = \tau(!\{Q\}) = \tau(!(\{Q\}) \equiv \tau(P\sigma). \\
\blacktriangle P & \equiv P_1 \mid P_2. \text{ Then } \tau(P)\sigma \equiv \tau(P_1 \mid P_2)\sigma = (\tau(P_1) \mid \tau(P_2))\sigma = \tau(P_1)\sigma \mid \tau(P_2)\sigma \text{ (by induction)} = \tau(P_1)\sigma \mid \tau(P_2)\sigma = \tau(P_1 \mid P_2)\sigma = \tau((P_1 \mid P_2)\sigma) \equiv \tau(P\sigma). \\
\blacktriangle P & \equiv P_1 + P_2. \text{ Then } \tau(P)\sigma \equiv \tau(P_1 + P_2)\sigma = (\tau(P_1) + \tau(P_2))\sigma = \tau(P_1)\sigma + \tau(P_2)\sigma \text{ (by induction)} = \tau(P_1)\sigma + \tau(P_2)\sigma = \tau((P_1 + P_2)\sigma) \equiv \tau(P\sigma).
\end{align*}\]
\[ P \equiv Q \setminus z. \] Then \( T(P)\sigma \equiv T(Q)'z\sigma = (w)T_{wz}(Q)\sigma = (w)T_{wz}(Q)\sigma. \) It suffices now to show that for any \( P T_{wz}(P)\sigma = T_{wz}(P)\sigma \) if \( s(w) = w \) given \( T(P)\sigma = T(P)\sigma \) by the induction hypothesis.

In this case, \( T(P)\sigma = (w)T_{wz}(Q)\sigma \equiv (w)T_{wz}(Q)\sigma = T(Q)\sigma \equiv T((Q)'z\sigma) \equiv T(P)\sigma \) as desired.

The proof is by induction on \( P \)'s formation.

\[ P \equiv 0. \] Then \( T_{wz}(P)\sigma \equiv T_{wz}(0)\sigma = 0 \sigma = T_{wz}(0)\sigma = T_{wz}(P)\sigma. \)

\[ P \equiv \tau. \] Then \( T_{wz}(P)\sigma \equiv T_{wz}(\tau. Q)\sigma = (\tau. T_{wz}(Q))\sigma = \tau. T_{wz}(Q)\sigma = \tau. T_{wz}(Q)\sigma \) (by induction) \( = T_{wz}(\tau. Q)\sigma = T_{wz}(Q)\sigma = T_{wz}(P)\sigma. \)

\[ P \equiv x. \] Then \( T_{wz}(P)\sigma \equiv T_{wz}(x. Q)\sigma = \}

\[ P \equiv !Q. \] Then \( T_{wz}(P)\sigma \equiv T_{wz}(!Q)\sigma = (!T_{wz}(Q))\sigma = !T_{wz}(Q)\sigma = !T_{wz}(Q)\sigma \) (by induction) \( = T_{wz}((!Q)\sigma) = T_{wz}(P)\sigma. \)

\[ P \equiv [x = y]Q. \] Then \( T_{wz}(P)\sigma \equiv T_{wz}([x = y]Q)\sigma = ([x = y]T_{wz}(Q))\sigma = [x = y][x = y]T_{wz}(Q)\sigma \) (by induction) \( = T_{wz}([x = y]Q)\sigma = T_{wz}(P)\sigma. \)

\[ P \equiv [x \neq y]Q \) is similar.

\[ P \equiv (P_1 + P_2). \] Then \( T_{wz}(P)\sigma \equiv T_{wz}(P_1 + P_2)\sigma = (T_{wz}(P_1) + T_{wz}(P_2))\sigma = T_{wz}(P_1)\sigma + T_{wz}(P_2)\sigma \) (by induction) \( = T_{wz}(P_1)\sigma + T_{wz}(P_2)\sigma = T_{wz}(P)\sigma. \)

\[ P \equiv P_1 | P_2. \] Then \( T_{wz}(P)\sigma \equiv T_{wz}(P_1 | P_2)\sigma = (T_{wz}(P_1) | T_{wz}(P_2))\sigma = T_{wz}(P_1)\sigma | T_{wz}(P_2)\sigma \) (by induction) \( = T_{wz}(P_1)\sigma | T_{wz}(P_2)\sigma = T_{wz}(P)\sigma. \)

So far we have proved that, under the assumptions mentioned above, for any agent \( P \) without occurrences of the blocking operator, \( T_{wz}(P)\sigma = T_{wz}(P)\sigma \). This fact will be used below.

\[ P \equiv Q \setminus z'. \] Then \( T_{wz}(P)\sigma \equiv T_{wz}(Q \setminus z')\sigma = T_{wz}(T(Q \setminus z')\sigma = T_{wz}(T(Q \setminus z')\sigma \) (since \( T(Q \setminus z')\) does not contain occurrences of blocking) \( = T_{wz}(T(Q \setminus z')\sigma \) (by induction) \( = T_{wz}(T(Q \setminus z')\sigma) = T_{wz}(Q \setminus z')\sigma = T_{wz}(P)\sigma. \)
Lemma 2.10: For any agent $P$, if $\tau_{\varphi_z}(P) \xrightarrow{\varphi_y} Q$, then $y \neq w$. Moreover, if $\tau_{\varphi_z}(P) \xrightarrow{\varphi} Q$, then $z \not\in s(\alpha)$.

Proof: An easy structural induction on $P$’s formation. This illustrates the idea of the transformation $\tau_{\varphi_z}$, which is to substitute a new fresh name for $z$ in internal communications, but to disallow such communication with the environment (by restricting $w$) while allowing $z$ to be exported (but extrusions of $w$ are not allowed).

Lemma 2.11: For any agent $P$, $\text{fn}(\tau(P)) \subseteq \text{fn}(P)$ and $\text{fn}(\tau_{\varphi_z}(P)) \subseteq \text{fn}(P) \cup \{w, z\}$.

Proof: An easy structural induction on $P$. The only interesting case is $\tau(P \setminus z)$, which reduces to $(w)(\tau_{\varphi_z}(P))$. If $P = 0$ then $z \not\in \tau(P \setminus z)$, and therefore we only have set inclusion, not equality.

Proposition 2.2: $P \simeq \tau(P)$ for any agent $P$.

Proof: Let $S = \{(\tau, \tau(P)) \mid P \text{ agent}\}$. Then $S$ is a strong bisimulation (up to $\alpha$-equivalence).

Assume first that $P \xrightarrow{\alpha} P'$. We have to prove that $\tau(P) \xrightarrow{\alpha} \tau(P')$, and also that if $\alpha \equiv x(y)$ and $y \not\in \text{fn}(P \tau(P))$, then $\tau(P')\{w/y\} = \tau(P'\{w/y\})$. But the latter follows from Lemma 2.9, and therefore we needn’t further consider the substitutions mentioned in point 2 of the definition of simulation.

Proof by induction on depth of inference.

TAU-ACT: $P \equiv \tau P' \alpha = \tau$. Then, by TAU-ACT $\tau(P) \equiv \tau(P') = \tau.\tau(P') \xrightarrow{\alpha} \tau(P)$.

OUTPUT-ACT: $P = \varphi y. P', \alpha = \varphi y$. Then, by OUTPUT-ACT $\tau(P) \equiv \tau(\varphi y. P') = \varphi y.\tau(P') \xrightarrow{\alpha} \tau(P')$.

INPUT-ACT: $P = x(y).Q, \alpha = x(y')$, $y' \not\in \text{fn}(y)Q$, $P' = Q[y'/y]$. Then, $\tau(P) \equiv \tau(x(y).Q) = x(y).\tau(Q) \xrightarrow{x(y')} \tau(Q)\{y'/y\} = \tau(Q[y'/y])$ (by lemma 2.9) $= \tau(P')$.

REP: $P \equiv !Q$. By a shorter inference, $Q !Q \xrightarrow{\alpha} P'$. By induction, $\tau(Q) !Q = !\tau(Q) \xrightarrow{\alpha} \tau(P')$. Then, by REP, $\tau(P) \equiv !\tau(Q) \xrightarrow{\alpha} \tau(P')$.

SUM: $P \equiv P_1 + P_2$. Then, by a shorter inference, $P_1 \xrightarrow{\alpha} P'$.

By induction, $\tau(P_1) \xrightarrow{\alpha} \tau(P')$. Then, by SUM $\tau(P) \equiv \tau(P_1 + P_2) = \tau(P_1) + \tau(P_2) \xrightarrow{\alpha} \tau(P')$.

MATCH: $P \equiv [x = x]Q$. Then, by a shorter inference $Q \xrightarrow{\alpha} P'$. By induction, $\tau(Q) \xrightarrow{\alpha} \tau(P')$ Then, by MATCH

$\tau(P) \equiv \tau([x = x]Q) = [x = x]\tau(Q) \xrightarrow{\alpha} \tau(P')$.

MISMATCH: is similar.

PAR: $P \equiv P_1 \mid P_2$. Then, $P' \equiv P'^{P_1} | P_2$ and by a shorter inference

$P_1 \xrightarrow{\alpha} P'. \text{ By induction, } \tau(P_1) \xrightarrow{\alpha} \tau(P')$.

By assumption $\text{fn}(\alpha) \cap \text{fn}(\tau(P_2)) = \emptyset$. Then, by PAR $\tau(P) \equiv \tau(P_1 | P_2) = \tau(P_1) | \tau(P_2) \xrightarrow{\alpha} \tau(P'') | \tau(P_2) = \tau(P'' | P_2) = \tau(P')$.

COM: $P \equiv P_1 | P_2$. Then, $P' \equiv P'_1 | P'_2[y/z]$, $\alpha \equiv \tau$, and by a shorter inference $P_1 \xrightarrow{\varphi_y} P'_1, P_2 \xrightarrow{\varphi} P'_2$. By induction, $\tau(P_1) \xrightarrow{\varphi_y} \tau(P'_1)$,
\( T(P_1) \xrightarrow{\alpha} T(P_2) \). Then, by \COM, \( T(P) \equiv T(P_1 \mid P_2) = T(P_1) \mid T(P_2) \xrightarrow{\alpha} T(P_2) \{ y/z \} = T(P_1) \mid T(P_2) \{ y/z \} \)
(by lemma 2.9) = \( T(P_1') \mid P_2 \{ y/z \} \equiv T(P') \).

**CLOSE:** \( P \equiv P_1 \mid P_2 \). Then, \( P' \equiv (w)(P_1' \mid P_2') \), \( \alpha \equiv \tau \), and by a shorter inference \( P_1 \xrightarrow{w} P_1', P_2 \xrightarrow{w} P_2' \). By induction,
\( T(P_1) \xrightarrow{w} T(P_1'), T(P_2) \xrightarrow{w} T(P_2') \). Then, by \CLOSE, \( T(P) \equiv T(P_1 \mid P_2) = T(P_1) \mid T(P_2) \xrightarrow{\alpha} (w)(T(P_1') \mid T(P_2')) = (w)(T(P_1') \mid P_2') \equiv T(P'). \)

**RES:** \( P \equiv (y)Q \). Then, \( P' \equiv (y)Q' \), \( y \not\in \text{fn}(\alpha) \), and by a shorter inference \( Q \xrightarrow{\alpha} Q' \). By induction, \( T(Q) \xrightarrow{\alpha} T(Q') \). Then, by \RES, \( T(P) \equiv (y)T(Q) \xrightarrow{\alpha} (y)T(Q') \equiv T(P') \).

**OPEN:** \( P \equiv (y)Q \). Then, \( P' \equiv Q \{w/y\} \), \( \alpha \equiv \pi(w) \) for some \( x \), and by a shorter inference \( Q \xrightarrow{\pi(w)} Q \). By induction, \( T(Q) \xrightarrow{\pi(w)} T(Q') \).
Since \( w \not\in \text{fn}(\text{fn}(y)Q') \), by lemma 2.11 \( w \not\in \text{fn}(T((y)Q')) = \text{fn}(y)T(Q')) \).
Then, by \OPEN, \( T(P) \equiv (y)T(Q) = (y)T(Q) \xrightarrow{\alpha} T(Q') \{w/y\} = (y)T(Q') \{w/y\} = (y)T(Q') \equiv T(P') \).

**BLOCK** \( P \equiv Q \| z \). Then, \( P' \equiv Q' \| z \), \( z \not\in \text{fn}(\alpha) \), and by a shorter inference \( Q \xrightarrow{\alpha} Q' \). Now \( T(P) \equiv T(Q \| z) = (w)T_{wz}(Q) \). We show by induction on the depth of inference that for any \( Q, Q' \) and fresh \( w \), and for any \( \alpha \) such that \( z \not\in \text{fn}(\alpha) \), if \( Q \xrightarrow{\alpha} Q' \), then \( T_{wz}(Q) \xrightarrow{\alpha} T_{wz}(Q') \), where \( \alpha = \pi(x, w(x)) \) or \( \pi(x) \) if \( \alpha = \pi(x), z(x) \) resp \( \pi(x) \), otherwise \( \alpha = \alpha \). By induction we may also assume that \( T(Q) \xrightarrow{\alpha} T(Q') \).

Then, since in our case \( z \not\in \text{fn}(\alpha) \), necessarily \( \alpha \equiv \alpha \) and thus \( T_{wz}(Q) \xrightarrow{\alpha} T_{wz}(Q') \). Hence, by \RES, \( T(Q \| z) \equiv (w)T_{wz}(Q) \xrightarrow{\alpha} (w)T_{wz}(Q') \) (since \( w \) is fresh and thus \( w \not\in \text{fn}(\alpha) \) = \( T(Q \| z) \equiv T(P') \) as desired.

**TAU-ACT:** \( Q \equiv \tau.Q', \: \alpha = \alpha = \tau \). Then, by \TAU-ACT, \( T_{wz}(Q) \equiv T_{wz}(\tau.Q') = \tau.T_{wz}(Q') \xrightarrow{\alpha} T_{wz}(Q') \).

**OUTPUT-ACT:** \( Q = \pi(y).Q', \: \alpha = \pi(y) \) if \( x = z \), otherwise \( \pi(y) \). Then,
\( T_{wz}(Q) \equiv T_{wz}(\pi(y).Q') = [x = z] \pi(y)T_{wz}(Q') + [x \neq z] \pi(y)T_{wz}(Q') \xrightarrow{\alpha} T_{wz}(Q') \).

**INPUT-ACT:** \( Q = x(y).Q', \: \alpha = x(y') \), \( Q' = Q'' \{ y'/y \} \), \( y' \not\in \text{fn}(y)Q'' \).
By lemma 2.11 and assumption \( z \not\in \text{fn}(\alpha) \) and \( w \) fresh, we get \( y' \not\in \text{fn}(y)T_{wz}(Q'') \).
Now, \( \alpha = x(y') \) if \( z = x \), \( \alpha = x(y') \) otherwise.
Then \( T_{wz}(Q) = (w)(x(y'), Q'') = [x = z]w(y')T_{wz}(Q'' \{ y'/y \}) + [x \neq z]x(y')T_{wz}(Q'' \{ y'/y \}) \xrightarrow{\alpha} T_{wz}(Q'' \{ y'/y \}) \{ y'/y'' \} \equiv T_{wz}(Q'' \{ y'/y'' \}) \equiv T_{wz}(Q'') \equiv T_{wz}(Q') \equiv T_{wz}(Q') \).

**REP:** \( Q \equiv !S \). Then, by a shorter inference \( S \mid S \xrightarrow{\alpha} Q' \). By induction,
\( T_{wz}(S \mid S) = T_{wz}(S) \mid T_{wz}(S) \xrightarrow{\alpha} T_{wz}(Q) \). Then, by \REP, \( T_{wz}(Q) = T_{wz}(S) \equiv !T_{wz}(S) \xrightarrow{\alpha} T_{wz}(Q) \).

**SUM:** \( Q \equiv Q_1 + Q_2 \). Then, by a shorter inference \( Q_1 \xrightarrow{\alpha} Q' \).
By induction, \( T_{wz}(Q_1) \xrightarrow{\alpha} T_{wz}(Q') \) and \SUM.
\[ T_{wz}(Q) \equiv T_{wz}(Q_1 + Q_2) = T_{wz}(Q_1) + T_{wz}(Q_2) \xrightarrow{\widetilde{\alpha}} T_{wz}(Q'). \]

**MATCH:** \( Q \equiv [x = x]Q'' \). Then, by a shorter inference \( Q'' \xrightarrow{\alpha} Q' \).

By induction \( T_{wz}(Q'') \xrightarrow{\alpha} T_{wz}(Q') \). Then, by MATCH
\[ T_{wz}(Q) \equiv T_{wz}([x = x]Q'') = [x = x]T_{wz}(Q'') \xrightarrow{\widetilde{\alpha}} T_{wz}(Q'). \]

**MISMATCH:** is similar.

**PAR:** \( Q \equiv Q_1 \mid Q_2 \). Then, \( Q' \equiv Q'' \mid Q_2 \), \( \text{bn}(\alpha) \cap \text{fn}(Q_2) = \emptyset \)
and by a shorter inference \( Q_1 \xrightarrow{\alpha} Q'' \). By induction
\[ T_{wz}(Q_1) \xrightarrow{\alpha} T_{wz}(Q''), \] Hence, since by lemma 2.11 and assumption \( z \notin \text{bn}(\alpha) = \text{bn}(\alpha) \cap \text{fn}(T_{wz}(Q_2)) = \emptyset \), and thus by PAR
\[ T_{wz}(Q) \equiv T_{wz}(Q_1 \mid Q_2) = T_{wz}(Q_1) \mid T_{wz}(Q_2) \xrightarrow{\alpha} T_{wz}(Q'') \mid T_{wz}(Q_2) = T_{wz}(Q'' \mid Q_2) \equiv T_{wz}(Q'). \]

**COM:** \( Q \equiv Q_1 \mid Q_2 \). Then, for some \( x, y, y' \), \( Q' \equiv Q'_1 \mid Q'_2 \{y/y'\}, \alpha = \hat{\alpha} = \tau \), and by shorter inferences \( Q_1 \xrightarrow{\alpha} Q'_1, Q_2 \xrightarrow{\alpha_2} Q'_2, \alpha_1 = \varphi(y), \alpha_2 = x(y') \).

By induction \( T_{wz}(Q_1) \xrightarrow{\hat{\alpha}} T_{wz}(Q'_1), T_{wz}(Q_2) \xrightarrow{\alpha_2} T_{wz}(Q'_2) \). Then, by COM
\[ T_{wz}(Q) \equiv T_{wz}(Q_1 \mid Q_2) = T_{wz}(Q_1) \mid T_{wz}(Q_2) \xrightarrow{\hat{\alpha}} T_{wz}(Q'_1) \mid T_{wz}(Q'_2) \{y/y'\} = (y)(T_{wz}(Q'_1 \mid Q'_2)) \equiv (y)(T_{wz}(Q'_1, Q'_2)) \equiv T_{wz}(Q'). \]

**CLOSE:** \( Q \equiv (y)S \). Then, \( Q' \equiv (y)S' \), \( y \notin n(\alpha) \) and by a shorter inference \( S \xrightarrow{\alpha} S' \). By induction, \( T_{wz}(S) \xrightarrow{\alpha} T_{wz}(S') \). Since by assumption \( w \) is fresh and \( y \notin n(\alpha) \), then \( y \notin n(\alpha) \). Thus, by RES \[ T_{wz}(Q) \equiv T_{wz}(y)S \equiv (y)(T_{wz}(S)) \xrightarrow{\alpha} (y)(T_{wz}(S')) \equiv T_{wz}(Q'). \]

**RES:** \( Q \equiv (y)S \). Then, \( Q' \equiv S' \{y'/y\}, \alpha \equiv \varphi(y') \), and by a shorter inference \( S \xrightarrow{\beta} S' \), \( y \neq x, y' \notin n((y)S') \). By induction, \( T_{wz}(S) \xrightarrow{\beta} T_{wz}(S') \), where \( \beta = \varphi(y) \) if \( x = z \), otherwise \( \beta = \varphi \). Since by assumption \( y' \notin n(\alpha) = \{z\} \) and \( w \) is fresh, by lemma 2.11 \( y' \notin n((y)T_{wz}(S')) \). In any case, by OPEN
\[ T_{wz}(Q) = T_{wz}(y)S \equiv (y)(T_{wz}(S)) \xrightarrow{\alpha} T_{wz}(S') \{y'/y\} = (y)(T_{wz}(S')) \xrightarrow{\alpha} T_{wz}(Q'). \]

Note that so far we have proved the claim for agents without occurrences of the blocking operator

**BLOCK:** \( Q \equiv S \backslash z' \). Then \( Q' = S' \backslash z' \), \( z \notin n(\alpha) \), and by a shorter inference \( S \xrightarrow{\alpha} S' \). Now, \( T_{wz}(Q) = T_{wz}(S \backslash z') = T_{wz}(T(S \backslash z')) = T_{wz}(T(Q)) \). Since \( T(Q) \xrightarrow{\alpha} T(Q') \) by induction, and noting that the blocking operator does not occur in \( T(Q) \), by the remark above we get \( T_{wz}(Q) \xrightarrow{\widetilde{\alpha}} T_{wz}(Q') \).
Now we have to show the other direction.

Assume $\mathcal{T}(P) \xrightarrow{\alpha} Q'$. We must show that there is an agent $Q$ such that
\[ P \xrightarrow{\alpha} Q \quad \text{and} \quad (Q, Q') \in S, \quad \text{that is,} \quad Q' = \mathcal{T}(Q). \]

The proof is by induction on the length of inference of $\mathcal{T}(P) \xrightarrow{\alpha} Q'$.

**TAU-ACT:** Then $P \equiv \tau.Q$, $\mathcal{T}(P) \equiv \mathcal{T}(\tau.Q) = \tau.\mathcal{T}(Q) \xrightarrow{\tau} \mathcal{T}(Q) \equiv Q'$. Then, by TAU-ACT $P \equiv \tau.Q \xrightarrow{\alpha} Q$, and $\mathcal{T}(Q) \equiv Q'$.

**OUTPUT-ACT:** Then $P \equiv \overline{x\cdot y}.Q$ for some $x, y$, $\alpha \equiv \overline{x\cdot y}$, $\mathcal{T}(P) \equiv \mathcal{T}(\overline{x\cdot y}.Q) \equiv \mathcal{T}(\overline{x\cdot y}).\mathcal{T}(Q) \equiv Q'$. Then, by OUTPUT-ACT $P \equiv \overline{x\cdot y}.Q \xrightarrow{\alpha} Q$, and $\mathcal{T}(Q) \equiv Q'$.

**INPUT-ACT:** Then $P \equiv x(y).P'$ for some $x, y$, $\mathcal{T}(P) \equiv \mathcal{T}(x(y).P') = x(y).\mathcal{T}(P') \xrightarrow{x(y)' \equiv P'} \mathcal{T}(P') \{y' / y\} = \mathcal{T}(P\{y' / y\})$ (by lemma 2.9) $= Q'$, $y' \notin \text{fn}(y)$. By definition of simulation we may assume $y' \notin n(\mathcal{T}(P), P)$. Then, $y' \notin n(P) \equiv n(x(y).P')$, hence $y' \notin \text{fn}(y)P')$.

Then, by INPUT-ACT $P \equiv x(y).P' \xrightarrow{x(y)'} P' \{y' / y\} \equiv Q$, and $\mathcal{T}(Q) = Q'$.

**SUM:** Then $P \equiv P_1 + P_2$, $\mathcal{T}(P) \equiv \mathcal{T}(P_1 + P_2) = \mathcal{T}(P_1) + \mathcal{T}(P_2) \xrightarrow{\alpha} Q'$, and by a shorter inference $\mathcal{T}(P_1) \xrightarrow{\alpha} Q$, $Q' = \mathcal{T}(Q)$. Hence, by SUM $P \equiv P_1 + P_2 \xrightarrow{\alpha} Q$, $\mathcal{T}(Q) = Q'$.

**MATCH:** Then $P \equiv [x = x]P'$ for some $x$, $\mathcal{T}(P) \equiv \mathcal{T}([x = x]P') = [x = x] \mathcal{T}(P') \xrightarrow{\alpha} Q'$, and by a shorter inference $\mathcal{T}(P') \xrightarrow{\alpha} Q'$.

Hence, by MATCH $P \equiv [x = x]P' \xrightarrow{\alpha} Q$, and $\mathcal{T}(Q) = Q'$.

**MISMATCH:** is similar.

**REP:** Then $P \equiv ![P']$, $\mathcal{T}(P) \equiv ![\mathcal{T}(P')] \equiv ![\mathcal{T}(P')] \xrightarrow{\alpha} Q'$, and by a shorter inference $\mathcal{T}(P') \xrightarrow{\alpha} Q'$. By induction, $P' \xrightarrow{\alpha} Q$, $Q' = \mathcal{T}(Q)$. Hence, by REP $![P'] \xrightarrow{\alpha} Q$, and $\mathcal{T}(Q) \equiv Q'$.

**PAR:** Then $P \equiv (P_1 | P_2)$, $\mathcal{T}(P) \equiv \mathcal{T}(P_1 | P_2) = \mathcal{T}(P_1) | \mathcal{T}(P_2) \xrightarrow{\alpha} P' | \mathcal{T}(P_2) \equiv Q'$, for some agent $P'$, and by a shorter inference $\mathcal{T}(P_1) \xrightarrow{\alpha} P'$, $\text{bn}(\alpha) \cap \text{fn}(\mathcal{T}(P_2)) = \emptyset$.

We may also assume by definition of simulation that $\text{bn}(\alpha) \cap \text{fn}(P_2) = \emptyset$. By induction $P_1 \xrightarrow{\alpha} P''$, $P' = \mathcal{T}(P'')$.

Then, by PAR $P \equiv P_1 | P_2 \xrightarrow{\alpha} P'' | P_2 = Q$, and $\mathcal{T}(Q) \equiv \mathcal{T}(P'') | P_2 = Q'$.

**COM:** Then $P \equiv P_1 | P_2$, $\mathcal{T}(P) \equiv \mathcal{T}(P_1 | P_2) = \mathcal{T}(P_1) | \mathcal{T}(P_2) \xrightarrow{\tau} Q'_1 | Q'_2 \{y' / y\} = Q'$. Then, by shorter inferences $\mathcal{T}(P_1) \xrightarrow{\overline{y}} Q'_1$, $\mathcal{T}(P_2) \xrightarrow{x(y)'} Q'_2$.
By induction, \( P_1 \xrightarrow{\tau y} Q_1 \), \( P_2 \xrightarrow{\tau (y')} Q_2 \), where \( Q'_1 = T(Q_1) \) and \( Q'_2 = T(Q_2) \).
Hence, by \( \text{CLOSE} \) \( P \equiv P_1 \parallel P_2 \), \( P_1 \xrightarrow{\alpha} Q_1 \parallel P_2 \xrightarrow{\alpha} Q'_2 \), and
\( T(Q) = T(Q_1 \parallel Q_2 [y/y']) = T(Q_1) \parallel T(Q_2 [y/y']) = (\text{by lemma 2.9}) \)
\( T(Q_1) \parallel T(Q_2 [y/y']) = Q'_1 \parallel Q'_2 \equiv Q' \).

**Open :** Then \( P \equiv P_1 \parallel P_2 \), \( T(P) \equiv T(P_1 \parallel P_2) \xrightarrow{\alpha} (w)(Q'_1 \parallel Q'_2) = Q' \),
\( \alpha = \tau \). Then, by shorter inferences \( T(P_1) \xrightarrow{\tau y} Q'_1 \), \( T(P_2) \xrightarrow{\tau (y')} Q'_2 \).
By induction, \( P_1 \xrightarrow{\tau y} Q_1 \), \( P_2 \xrightarrow{\tau (y')} Q_2 \), where \( Q'_1 = T(Q_1) \) and \( Q'_2 = T(Q_2) \).
Hence, by \( \text{CLOSE} \) \( P \equiv P_1 \parallel P_2 \xrightarrow{\alpha} (w)(Q_1 \parallel Q_2) = Q \), and \( T(Q) = T((w)(Q_1 \parallel Q_2)) = (w)(T(Q_1) \parallel T(Q_2)) \equiv Q' \).

\( \text{RES} : \) Here we have two alternatives: either \( P \) has form \( (y)P' \) or \( S \parallel z \).
Suppose \( P \equiv (y)P' \) for some agent \( P' \).
Then, \( T(P) \equiv T((y)P') = (y)T(P') \xrightarrow{\alpha} (y)Q'' \equiv Q' \), \( y \notin n(\alpha) \).
By a shorter inference \( T(P') \xrightarrow{\alpha} Q'' \). By induction,
\( P' \xrightarrow{\alpha} P'', \ Q'' = T(P'') \). Then, since \( y \notin n(\alpha) \), by \( \text{RES} \) \( P \equiv (y)P' \xrightarrow{\alpha} (y)P'' = Q \). Now, \( T(Q) \equiv T((y)P'') = (y)T(P'') = (y)Q'' = Q' \).

Now assume \( P \equiv S \parallel z \) for some agent \( S \).
Then \( T(P) = (w)T_{uz}(S) \xrightarrow{\alpha} (w)Q'' = Q' \), and by a shorter inference \( T_{uz}(S) \xrightarrow{\alpha} Q'' \), \( w \notin n(\alpha) \). We may also assume that \( z \notin n(\alpha) \).

It is enough to show that for any \( P,Q \), if \( T_{uz}(P) \xrightarrow{\beta} Q \) then there is a \( P' \)
such that \( P \xrightarrow{\beta \sigma} P' \), where \( \sigma = \{z/w\} \) and \( T_{uz}(P') \equiv_a Q \).

In this case, since \( w \notin n(\alpha) \), then \( \alpha \sigma \equiv \alpha \), hence \( S \xrightarrow{\alpha} S' \) for some \( S' \) such that \( T_{uz}(S') = Q'' \).
Hence, by lemma 2.11 \( z \notin n(\alpha) \), thus \( z \notin n(\alpha) \) and then by \( \text{BLOCK} \)
\( S \parallel z \xrightarrow{\alpha} S' \parallel z \) and \( T(S' \parallel z) = (w)T_{uz}(S') = (w)Q'' \) as desired.

Proof by induction on depth of inference:

**TAU-\text{ACT} :** Then \( P \equiv \tau \cdot P' \), \( \beta \equiv \tau \), \( T_{uz}(P) \equiv T_{uz}(\tau \cdot P') = \tau \cdot T_{uz}(P') \xrightarrow{\beta} T_{uz}(P') = Q \). Hence \( P \equiv \tau \cdot P' \xrightarrow{\beta \sigma} P' \), \( T_{uz}(P') = Q \).

**OUTPUT-\text{ACT} :** impossible by definition of \( T_{uz} \).
INPUT-ACT: impossible by definition of $\tau_{yz}$.

SUM: Here we have three alternatives:

1. $P \equiv P_1 + P_2$. Then, $\tau_{yz}(P) \equiv \tau_{yz}(P_1 + P_2) = 
   \tau_{yz}(P_1) + \tau_{yz}(P_2) \to Q$ and by a shorter inference $\tau_{yz}(P_1) \to Q$. 
   By induction, $P_1 \to P_1'$ and $\tau_{yz}(P_1') = Q$. 
   Then, by SUM $P \equiv P_1 + P_2 \to P_1'$ and $\tau_{yz}(P_1') = Q$.

2. $P \equiv \tau_{xy} P'$. Then, $\tau_{yz}(P) \equiv \tau_{yz}(\tau_{xy} P') = 
   [x = z] \tau_{xy} \tau_{yz}(P') \to [x \neq z] \tau_{xy} \tau_{yz}(P') \to Q$. 
   Assume $x = z$. Then we must have by a shorter inference $\tau_{xy} \tau_{yz}(P') \to \tau_{yz}(P') \equiv Q$. 
   By a shorter inference $\tau_{xy} \tau_{yz}(P') \to \tau_{yz}(P') \equiv Q$, $\beta \equiv \tau_{xy}$.

   On the other hand, if $x \neq z$ then we must have by a shorter inference $\tau_{xy} \tau_{yz}(P') \to \tau_{yz}(P') = Q$, $\beta \equiv \tau_{xy}$.
   In either case $[x = z] \tau_{xy} \tau_{yz}(P') \to [x \neq z] \tau_{xy} \tau_{yz}(P') \to Q$.
   Then, by OUTPUT-ACT $P \equiv \tau_{xy} P' \to P'$ and $\tau_{yz}(P') = Q$.

3. $P \equiv x(y) P''$. Then, $\tau_{yz}(P) \equiv \tau_{yz}(x(y) P'') = 
   [x = z] w(y). \tau_{yz}(P'') + [x \neq z] x(y). \tau_{yz}(P'') \to Q$. Assume $x = z$.
   Then by a shorter inference $w(y). \tau_{yz}(P'') \equiv Q$. 
   On the other hand, if $x \neq z$, then by a shorter inference $x(y). \tau_{yz}(P'') \equiv Q$. 
   Then, in either case $\beta \equiv x(y)$.

   Thus by INPUT-ACT $P \equiv x(y) P'' \to P''$, $\tau_{yz}(P'') = Q$. Then, by MATCH, 
   $P \equiv [x = x] P'' \to P''$, $\tau_{yz}(P'') = Q$.

MATCH: Then $P \equiv [x = x] P''$, $\tau_{yz}(P) \equiv \tau_{yz}([x = x] P'') = 
   [x = x] \tau_{yz}(P'') \to Q$ and by a shorter inference $\tau_{yz}(P'') \to Q$.

By induction $P'' \to P'$, $\tau_{yz}(P') = Q$. Then, by MATCH, 
$P \equiv [x = x] P'' \to P'$, $\tau_{yz}(P') = Q$.

MISMATCH: similar.

REP: Then $P \equiv [P'' ||] P''$, $\tau_{yz}(P) \equiv \tau_{yz}([P'' ||] P'') = 
   [P'' ||] \tau_{yz}(P'') \to Q$, for some agent $P''$, and by a shorter inference $\tau_{yz}(P'') \to Q$.

By induction, $P'' || P'' \to R$ for some agent $R$ 
such that $\tau_{yz}(R) = P''$. Then, by REP $P \equiv [P'' || P'' \to R = P'$ and 
$\tau_{yz}(P') \equiv \tau_{yz}(R) = P'' = Q$.

PAR: Then $P \equiv P_1 | P_2$, $\tau_{yz}(P) \equiv \tau_{yz}(P_1 | P_2) = 
   \tau_{yz}(P_1) | \tau_{yz}(P_2) \to R \tau_{yz}(P_2) = Q$, $\beta \equiv \emptyset$.

Then, by a shorter inference $\tau_{yz}(P_1) \to R$. By induction, 
$P_1 \to P_1'$, $\tau_{yz}(P_1') = R$. Assuming $\beta \equiv \emptyset$, by PAR 
$P \equiv P_1 | P_2 \to P_1'$ and $\tau_{yz}(P') \equiv \tau_{yz}(P_1') | P_2 = 
   \tau_{yz}(P_1') | \tau_{yz}(P_2) = R \tau_{yz}(P_2) = Q$. 

219
The proof of the theorem is now complete.
A.2 Appendix to section 3.3

We call a channel \( x \) in \( \pi B \) an *spawning channel* if it corresponds to an agent variable \( X \) in \( \Pi B \) or any channel \( v \) in \( \pi B \) that may be used as a pointer to a spawning process \( \text{spawn}_w(F) \) for any \( F \in \Pi B \). Also, we call a channel \( v \in \pi B \) a *sender channel* if it occurs in the context of an agent \( \text{send}_v(P) \) for any \( P \in \Pi B \).

The following remarks may be checked easily:

**Remark 1:** A spawning channel \( w \) may be the subject of an input in \( T(P) \) only in the context of an agent of type \( \text{spawn}_w(F) \).

**Remark 2:** Only spawning and restricted sender channels may be the subject of any actions executed by \( \text{spawn}_w(F) \), for any \( F \in \Pi B \). Likewise, only channel \( v \) may be the subject of any actions executed by \( \text{send}_v(P) \), for any \( P \in \Pi B \).

**Remark 3:** If both \( \text{spawn}_w(F) \) and \( \text{spawn}_w(G) \) occurs in \( T(P) \), for any agent \( P \in \Pi B \), then either \( F \equiv G \) or each occurrence of the spawning channel \( w \) is bound by different restrictions (\( w \)). In other words, one and the same “pointer” may not point to different agents.

**Remark 4:** Spawning and sender channels never occurs in matching or blocking.

We extend the definition of \( T \) for terms in \( \Pi B \) to static contexts in \( \Pi B \) by letting \( T[.] = [.] \). We have the following:

**Lemma 3.1:** For any process \( P \in \Pi B \) and any static context \( C[.] \), \( T(C[P]) = T(C)[T(P)] \).

**Proof:** By induction on \( C \)'s formation.

- \( C \equiv [.] \). Then
  \[ T(C[P]) = T(P) = T(C)[T(P)] \] (since \( T[.] = [.] \)).

- \( C \equiv Q \). Then
  \[ T(C[P]) = T(Q[P]) = T(Q) = T(Q)[T(P)] = T(C)[T(P)]. \]

- \( C \equiv C_1 \mid C_2 \). Then
  \[ T(C[P]) = T((C_1 \mid C_2)[P]) = T(C_1[P] \mid C_2[P]) = T(C_1[P]) \mid T(C_2[P]) = T(C_1)[T(P)] \mid T(C_2)[T(P)] \] (by induction) = \( (T(C_1) \mid T(C_2))[T(P)] = T(C_1 \mid C_2)[T(P)] = T(C)[T(P)] \).

- \( C \equiv (x)C' \). Then
  \[ T(C[P]) = T(((x)C')[P]) = T((x)(C'[P])) = (x)T(C'[P]) = (x)(T(C'))[T(P)] \] (by induction) = \( ((x)T(C'))[T(P)] = T((x)C')[T(P)] = T(C')[T(P)]. \)

- \( C \equiv !C' \). \( C \equiv C' \setminus \{\} \). Similar.

**Lemma 3.2** For closed \( P, P \downarrow_{w} \iff T(P) \downarrow_{w} \).

**Proof:** An easy induction on the structure of \( P \).

**Lemma 3.3** Let \( P, Q \in \pi B \) be transformations of agents in \( \Pi B \) or any derivative of such agents, such that for channel \( w \) \( \text{spawn}_w(G) \) does not occur in either \( P \) or \( Q \) for any agent \( G \in \Pi B \). Then

(i) \( (w)(\text{spawn}_w(F) \mid P) \mid \text{spawn}_w(F) \sim_r P \mid \text{spawn}_w(F) \)

(ii) \( (w)(\text{spawn}_w(F) \mid P) \mid Q \sim_r (w)(\text{spawn}_w(F) \mid P) \mid (w)(\text{spawn}_w(F) \mid Q) \)
(iii) \((w)(\text{spawn}_w(F) | P) \sim_r !(w)(\text{spawn}_w(F) | P)\)

(iv) \((w)(P | \text{spawn}_w(F)) + (w)(Q | \text{spawn}_w(F)) \sim_r (w)((P + Q) | \text{spawn}_w(F))\).

**Proof of (i)** We make below extensive use, without mention, of lemmas 2.3 and 2.4, and the remarks at the beginning of this appendix.

Let \(B\) be a relation consisting of all pairs of form

\[(\bar{y}u')(\prod w | \text{spawn}_{w'}(F) | P | \text{spawn}_w(F))\]

for all reduced compositions \(\prod\) and name vectors \(\bar{y}\), such that neither \(\text{spawn}_w(G)\) nor \(\text{spawn}_{w'}(G)\) occur in \(\prod\) or \(P\) for any abstraction \(G \in \Pi B\). This implies that both \(\prod \xrightarrow{w(u)}\) and \(\prod \xrightarrow{w'(u)}\) are impossible, as well as \(P \xrightarrow{w(u)}\) and \(P \xrightarrow{w'(u)}\).

We want to show that \(B\) is a strong barbed bisimulation in \(\pi B\) (up to strong equivalence). In this case the result follows since, assuming \(w'\) is fresh, for any reduced composition \(\prod\) we have

\[(\bar{y})(\prod w | (w)(\text{spawn}_w(F) | P) | \text{spawn}_w(F)) \sim \]

\[(\bar{y})(\prod w' | (w')(\text{spawn}_{w'}(F) | P^P | \text{spawn}_w(F)) \sim \]

\[(\bar{y})(\prod w' | (Pw' | \text{spawn}_w(F)) \sim_r \]

\[(\bar{y})(\prod w' | P^P | \text{spawn}_w(F))\]

Suppose first that

\[(\bar{y}u')(\prod w | \text{spawn}_{w'}(F) | P | \text{spawn}_w(F)) \xrightarrow{\tau} Q\]

1. If the \(\tau\)-action was a result of an internal action by \(P\), and

\[P \xrightarrow{\tau} P'\]

then

\[Q \sim (\bar{y}u')(\prod w | \text{spawn}_{w'}(F) | P' | \text{spawn}_w(F)),\]

and since neither \(\text{spawn}_w(G)\) nor \(\text{spawn}_{w'}(G)\) occur in \(P\) for any \(G\), the \(\tau\)-transition must have taken place through a channel \(x\) such that \(x \neq w, x \neq w'\) and thus

\[P\{w/u'\} \xrightarrow{\tau} P^P\{w/u'\}\]

Hence

\[(\bar{y})(\prod w' | P^P | \text{spawn}_w(F)) \xrightarrow{\tau} R = \]

\[(\bar{y})(\prod w' | P^P | \text{spawn}_w(F))\]

and

\[(Q, R) \in B.\]

2. if the \(\tau\)-action was a result of an internal action by \(\prod\), and \(\prod \xrightarrow{\tau} \prod'\), then the \(\tau\)-transition, as in case 1, must have taken place through a channel \(x\) such that \(x \neq w'\), and thus

\[\prod \xrightarrow{\tau} \prod'\]

and the result follows as in case 1.

3. If the action results from a synchronization between \(P\) and \(\text{spawn}_{w'}(F)\), then for some \(u\)

\[P \xrightarrow{u'(u)} P', \text{spawn}_{w'}(F) \xrightarrow{u'(u)} u(v)(u(x) \text{send}_w(F(x))) | \text{spawn}_{w'}(F)\]

222
and

\[ Q \sim (\bar{y}u')w'((\prod') | \text{spawn}_{w'}(F) | P' | \text{spawn}_{w}(F)) \]

where

\[ \prod' \sim \prod | u(v).u(x).send_w(F \langle x \rangle). \]

In this case we have

\[ P \{ w/u' \} \xrightarrow{\bar{y}u} P' \{ w/u' \} \]

and since

\[ \text{spawn}_w(F) \xrightarrow{w(u)} u(v).u(x).\text{send}_w(F \langle x \rangle) \]

we get

\[ (\bar{y})(\prod' \{ w/u' \} | P \{ w/u' \} | \text{spawn}_w(F)) \xrightarrow{\bar{y}u} R = (\bar{y}u)(\prod' \{ w/u' \} | P' \{ w/u' \} | \text{spawn}_w(F)) \]

as desired, since \((Q, R) \in B\).

4. If the action results from a synchronization between \(P\) and \(\text{spawn}_w(F)\), then for some \(u\)

\[ P \xrightarrow{w(u)} P', \text{spawn}_w(F) \xrightarrow{u} u(v).u(x).\text{send}_w(F \langle x \rangle) | \text{spawn}_w(F) \]

and

\[ Q \sim (\bar{y}u'w)(\prod' | \text{spawn}_{w'}(F) | P' | \text{spawn}_{w}(F)) \]

where

\[ \prod' \sim \prod | u(v).u(x).\text{send}_w(F \langle x \rangle). \]

In this case,

\[ P \{ w/u' \} \xrightarrow{\bar{y}u} P' \{ w/u' \} \]

and thus

\[ (\bar{y})(\prod' \{ w/u' \} | P \{ w/u' \} | \text{spawn}_w(F)) \xrightarrow{\bar{y}u} R = (\bar{y}u)(\prod' \{ w/u' \} | P' \{ w/u' \} | \text{spawn}_w(F)) \]

as desired, since \((Q, R) \in B\).

5. If the action results from a synchronization between \(\prod\) and \(\text{spawn}_w(F)\), resp \(\prod\)

and \(\text{spawn}_w(F)\), the argument is identical to cases 3 and 4 above if we substitute \(\prod\) for \(P\) and vice versa.

6. If the action results from a synchronization between \(P\) and \(\prod\), leading to \(P'\) and \(\prod'\), then, since a synchronization though \(w\) or \(w'\) is not possible, the synchronization must happen through a channel \(x\) such that \(x \neq w, x \neq w'\). In this case, if neither \(w\) nor \(w'\) is exchanged in the synchronization, the same synchronization may take place between \(\prod \{ w/u' \} \) and \(P \{ w/u' \} \), leading to \(\prod' \{ w/u' \} \) and \(P' \{ w/u' \} \), and the result follows easily.

Suppose then that \(u'\) is exchanged in the communication.

Assume

\[ P \xrightarrow{w} P', \prod \xrightarrow{w} \prod', Q = (\bar{y}u')(\prod' | \text{spawn}_{w'}(F) | P' | \text{spawn}_{w}(F)). \]

Then

\[ P \{ w/u' \} \xrightarrow{w} P' \{ w/u' \}, \prod \{ w/u' \} \xrightarrow{w} \prod' \{ w/u' \}, \]

\[ (\bar{y})(\prod \{ w/u' \} | P \{ w/u' \} | \text{spawn}_w(F)) \xrightarrow{\bar{y}u} R = \]

223
\[(\overline{y}(\prod \{w/w' \} | P^\prime \{w/w' \} | \text{spawn}_w(F)))\]

and \((Q, R) \in B\) as desired.

The case
\[\prod \overline{u} \rightarrow \prod', \ P \xrightarrow{\overline{u}} P',\]
is identical if we exchange \(\prod\) and \(P\) in the proof.

Assume now that \(w\) is exchanged in the communication.

Suppose
\[P \xrightarrow{\overline{u}} P', \ \prod \overline{u} \prod', \ Q = (\overline{y}u')(\prod' | \text{spawn}_{w'}(F) | P' | \text{spawn}_w(F)).\]

Then
\[P\{w/w' \} \xrightarrow{\overline{u}} P\{w/w' \}, \ \prod\{w/w' \} \xrightarrow{\overline{u}} \prod'\{w/w' \},\]
\[\overline{y}(\prod\{w/w' \} | P\{w/w' \} | \text{spawn}_w(F)) \xrightarrow{\overline{u}} R = \overline{y}(\prod\{w/w' \} | P'\{w/w' \} | \text{spawn}_w(F))\]

and \((Q, R) \in B\) as desired.

The case
\[\prod \overline{u} \rightarrow \prod', \ P \xrightarrow{\overline{u}} P',\]
is similar.

Suppose now that
\[(\overline{y})(\prod\{w/w' \} | \text{spawn}_w(F) | P\{w/w' \}) \xrightarrow{\overline{u}} Q.\]

1. If the transition is the result of an internal communication in \(P\), and
\[P\{w/w' \} \xrightarrow{\overline{u}} R,\]
than since it cannot be a synchronization along \(w\), we have
\[P \xrightarrow{\overline{u}} P'\text{ and } R \equiv P'\{w/w' \}.\]

But then
\[(\overline{y}u')(\prod | \text{spawn}_{w'}(F) | P | \text{spawn}_w(F)) \xrightarrow{\overline{u}} Q \sim (\overline{y}u')(\prod | \text{spawn}_w(F) | P' | \text{spawn}_w(F))\]

and \((Q, R) \in B\) as desired.

2. If the transition is the result of an internal communication in \(\prod\{w/w' \}\) the argument is similar.

3. If the transition result from a synchronization between \(P\{w/w' \}\) and \(\text{spawn}_w(F)\), then for some \(u\)
\[P\{w/w' \} \xrightarrow{\overline{u}} P'u', \ \text{spawn}_w(F) \xrightarrow{w(u)} u(v).u(x).\text{send}_v(F\langle x \rangle) | \text{spawn}_w(F),\]
and
\[R \equiv (\overline{y}u)(\prod' \{w/w' \} | P'u' | \text{spawn}_w(F))\]

where
\[\prod' \equiv \prod | u(v).u(x).\text{send}_v(F\langle x \rangle)\]

since
\[w' \not\in \text{fin}(u(v).u(x).\text{send}_v(F\langle x \rangle)).\]
We have necessarily two alternatives here:

Either \( P \xrightarrow{u(v)} P' \) or \( P \xrightarrow{w(u)} P' \) for some \( P' \) such that \( P' \{ w/u' \} = P'' \).

In either case

\[
(jy')(\prod \text{spawn}_{w'}(F) \mid P \mid \text{spawn}_{w}(F)) \xrightarrow{T} Q \equiv
(jy')(\prod \text{spawn}_{w}(F) \mid P \mid \text{spawn}_{w}(F)) \quad \text{and} \quad (Q, R) \in B.
\]

4. Synchronization between \( \prod \{ w/u' \} \) and \( \text{spawn}_{w}(F) \). Similar to case 3.

5. Synchronization between \( P\{w/u'\} \) and \( \prod \{ w/u' \} \).

The synchronization, say involving actions \( \alpha \) by \( P \) and \( \beta \) by \( \prod \), must take place along a channel \( x \neq w \), resulting in \( P'\{w/u'\} \) and \( \prod' \{ w/u' \} \) for some \( P' \) and \( \prod' \). But then a similar synchronization may take place between \( P \) and \( \prod \) involving actions \( \alpha \{w/u'\} \) resp. \( \beta \{w/u'\} \), resulting in \( P' \) resp. \( \prod' \), and the result follows.

**Proof of (ii):** From (i) we get, assuming \( w \) is fresh,

\[
(y)(\prod \text{spawn}_{w}(F) \mid P \mid Q) \sim
(y)(\prod \text{spawn}_{w}(F) \mid Q) \quad \text{(where} \quad \prod = \prod \mid P \text{)} \sim
((y)(\prod \text{spawn}_{w}(F) \mid Q) \mid \text{spawn}_{w}(F)) \sim
((y)(\prod \text{spawn}_{w}(F) \mid \text{spawn}_{w}(F)) \mid (w\{\text{spawn}_{w}(F) \mid Q\})) \sim
((y)(\prod \text{spawn}_{w}(F)) \mid \text{spawn}_{w}(F) \mid (w\{\text{spawn}_{w}(F) \mid Q\})).
\]

**Proof of (iii)** This lemma is a generalization of (ii). By (ii) we now that \( (w)(\text{spawn}_{w}(F) \mid P^n) \sim \text{ spawn}_{w}(F) \mid P^n \), where \( A^n \) is defined as the composition of \( A \) with itself \( n \) times. If some reduced context could distinguish between \( (w)(\text{spawn}_{w}(F) \mid !P) \) and \( !(w)(\text{spawn}_{w}(F) \mid P) \), then it may distinguish \( (w)(\text{spawn}_{w}(F) \mid P^n) \) and \( (w)(\text{spawn}_{w}(F) \mid P)^n \) for some \( n \), a contradiction.

**Proof of (iv)** Follows from the expansion rule.

**Lemma 3.4:** If \( X \) and \( Y \) are not process variables, then \( T(P)\{x/y\} = T(P\{x/y\}) \).

Otherwise \( T(P)\{x/y\} = T(P\{X/Y\}) \).

**Proof:** We show first that

(i) \( \text{spawn}_{w}(F)\{x/y\} = \text{spawn}_{w}(F\{x/y\}) \) if \( x, y \) are first-order, and

(ii) \( \text{spawn}_{w}(F)\{x/y\} = \text{spawn}_{w}(F\{X/Y\}) \) if \( x, y \) are higher-order.

In both cases we may assume that \( x, y \neq w \).

(i) If \( F = (x')P \) or \( F = (\lambda X')P \), then \( \text{spawn}_{w}(F) = w(u).u(w')P \).

In either case we may assume that \( x' \neq x \).

Thus it is enough to show that \( \text{send}_{w}(P)\{x/y\} = \text{send}_{w}(P\{x/y\}) \) for \( v \neq x, y \).

The proof is by induction on \( P \)'s formation. We may assume that \( x \) and \( y \) are distinct from \( w \) and from any of the special channels.

- \( P = 0 \). Then
  \[
  \text{send}_{w}(P)\{x/y\} = \text{send}_{w}(0)\{x/y\} = (\forall v. 0)\{x/y\} = \forall v. 0 = \text{send}_{w}(0) = \text{send}_{w}(0\{x/y\}) = \text{send}_{w}(P\{x/y\}).
  \]

- \( P = \overline{\nu}P' \). Then
  \[
  \text{send}_{w}(P)\{x/y\} = \text{send}_{w}(\overline{\nu}P')\{x/y\} = (\forall v. \overline{\nu}a. \overline{\nu}b. \text{send}_{w}(P'))\{x/y\} = \forall v. \forall a. \overline{\nu}b. \text{send}_{w}(P')\{x/y\} = \forall v. \forall a. \overline{\nu}b. \overline{\nu}a. \overline{\nu}b. \text{send}_{w}(P')\{x/y\} = (\text{by induction}) \text{send}_{w}(\overline{\nu}a. \overline{\nu}b. b)\{x/y\}. P'\{x/y\} = \text{send}_{w}(\overline{\nu}b. P')\{x/y\} = \text{send}_{w}(P\{x/y\}).
  \]

- \( P = \overline{\nu}(b). P' \), \( a(X') \), \( P' \) or \( \overline{\nu}(X') \). Similar.

- \( P = \overline{\nu}(F). P' \), \( F \) not a process variable. Then

\[\]
\[ \text{send}_r(P)(x/y) = \text{send}_e(\pi(F), P')\{x/y\} = \\
(\tau_0, \tau_0, \tau_0)(\text{send}_r(P) | \text{spawn}_w(F))\{x/y\} = \\
\tau_0(\text{send}_r(P)\{x/y\} | \text{spawn}_w(F)\{x/y\}) \\
\text{by induction} \\
= \text{send}_e(\pi(F), P')\{x/y\} = \text{send}_e(P\{x/y\}). \]

- \( P = P_1 + P_2 \). Then
  \[ \text{send}_r(P)(x/y) = \text{send}_e(P_1 + P_2)(x/y) = \\
(\tau_0, \tau_0, \tau_0)(\text{send}_e(P_1) | \text{send}_e(P_2))\{x/y\} = \\
\tau_0(\text{send}_e(P_1)\{x/y\} | \text{send}_e(P_2)\{x/y\}) = \\
\text{by induction} \\
= \text{send}_e(P_1\{x/y\} + P_2\{x/y\}) = \text{send}_e((P_1 + P_2)\{x/y\}) = \text{send}_e(P\{x/y\}). \]

- \( P = P_1 \mid P_2 \), \( P = [a = b]P' \), \( P = (x)P' \), \( P = !P' \), \( P = P' \setminus z \).

The proofs in all these cases are similar.

- \( P = Y'\{x'\} \). Then
  \[ \text{send}_r(P)(x/y) = \text{send}_e(Y'(x'))\{x/y\} = (\bar{y}(u), \tau_0, \tau_0)\{x/y\} = \\
\bar{y}(u)\{x/y\} = \text{send}_e(Y'(x')\{x/y\}) = \\
\text{send}_e((Y'(x'))\{x/y\}) = \text{send}_e(P\{x/y\}). \]

- \( P = Y'(X) \). Then
  \[ \text{send}_r(P)(x/y) = \text{send}_e(Y'(X))\{x/y\} = (\bar{y}(u), \tau_0, \tau_0)\{x/y\} = \\
\bar{y}(u)\{x/y\} = \text{send}_e(Y'(X)) = \\
\text{send}_e((Y'(X))\{x/y\}) = \text{send}_e(P\{x/y\}). \]

- \( P = Y'(F) \), \( F \) not a process variable. Then
  \[ \text{send}_r(P)(x/y) = \text{send}_e(Y'(F))\{x/y\} = (\bar{y}(u), \tau_0, \tau_0, \text{spawn}_w(F))\{x/y\} = \\
\bar{y}(u)\{x/y\} = \text{send}_e(Y'(F))\{x/y\} = \\
\text{send}_e((Y'(F))\{x/y\}) = \text{send}_e(P\{x/y\}). \]

(ii) The proof of the case \( \text{send}_e(P)(x/y) = \text{send}_e(P\{X/Y\}) \) is similar.

**Proof of lemma:** By induction on P’s formation.

Suppose \( x \) and \( y \) are first-order channels.

- \( P = 0 \). Then
  \[ T(P)(x/y) = T(0)(x/y) = 0\{x/y\} = 0 = T(0) = T(0|\{x/y\}). \]

- \( P = \alpha.P' \), \( \alpha \) a first-order synchronization. Then
  \[ T(P)(x/y) = T(\alpha.P')(x/y) = (\alpha, T(P'))\{x/y\} = \alpha\{x/y\}, T(P')\{x/y\} = \\
\alpha\{x/y\}, T(P')\{x/y\} \text{ (by induction)} = T(\alpha\{x/y\}, P'\{x/y\}) = \\
T(\alpha.P')(x/y) = T(P(x/y)). \]

- \( P = P_1 + P_2 \). Then
  \[ T(P)(x/y) = T(P_1 + P_2)(x/y) = (T(P_1) + T(P_2))\{x/y\} = \\
T(P_1)(x/y) + T(P_2)(x/y) = \\
T(P_1)(x/y) + T(P_2)(x/y) \text{ (by induction)} = \\
T(P_1)(x/y) + T(P_2)(x/y) = T((P_1 + P_2)(x/y)) = T(P(x/y)). \]

- \( P = P_1 | P_2 \), \( P = [a = b]P' \), \( P = (z)P' \), \( P = !P' \), \( P = P' \setminus z \).

All these cases are similar to the one above.

- \( P = a(X')P' \). Then \( x \neq x', y \neq x' \), since \( x \) and \( y \) are first-order.
  \[ T(P)(x/y) = T(a(X'), P')\{x/y\} = (a(x'), T(P'))\{x/y\} = \\
a(x')\{x'), T(P')(x/y) = a(x')\{x'). T(P(x/y)) \text{ (by induction)} = \\
T(a\{x/y\}(X')). P'\{x/y\} = T((a(X')). P')\{x/y\} = T(P(x/y)). \]
• $P = \pi(Y).P$. Similar

• $P = \pi(F).P'$, $F$ not a process variable. Then
  \[ T(P) = T(\pi(F).P')[x/y] = \pi(w) \cdot T(F)[x/y] = \pi[w](T(P) \mid spawn_w(F))[x/y]. \]

• $P = Y'(x')$, where $x \neq y$, $y \neq y'$. Then
  \[ T(P) = T(Y'(x'))[x/y] = Y'(\mu u. \alpha v. rec\langle v \rangle)[x/y] = T(Y'(x'))[x/y] = T(P[x/y]). \]

• $P = Y'(X')$. Similar.

• $P = Y'(F)$, where $x \neq y'$, $y \neq y'$, and $F$ is not a process variable. Then
  \[ T(P) = T(Y'(F))[x/y] = Y'(\mu u. \alpha v. (rec\langle v \rangle \mid spawn_w(F))[x/y] = T(Y'(F))[x/y] = T(P[x/y]). \]

The cases where $x$ and $y$ corresponds to high-order variables are similar. We give only the corresponding of the last one above.

• $P = Y'(F)$, $F$ not a process variable. Then
  \[ T(P) = T(Y'(F))[x/y] = Y'(\mu u. \alpha v. (rec\langle v \rangle \mid spawn_w(F))[x/y] = T(Y'(F))[x/y] = T(P[x/y]). \]

**Lemma 3.5:**
If $P \equiv_a Q$, then $T(P) \equiv_a T(Q)$.

**Proof:** An easy induction on $P$’s formation.

**Lemma 3.6:** For any closed agent $P \in \Pi B$, if $\mu$ is a first-order prefix, $\mu \neq \tau$, then

(i) if $P \xrightarrow{\mu} P'$ then $T(P) \xrightarrow{\mu} T(P')$;
(ii) if $T(P) \xrightarrow{\mu} Q$ then $P \xrightarrow{\mu} P'$ for some $P' \in \Pi B$ such that $T(P') = Q$.

**Proof:** by induction on the length of inference.

**• ALP:**
(i) Assume $P \xrightarrow{\mu} P'$ by ALP, $Q \xrightarrow{\mu} P'$, $Q \equiv_a P$.
Then, by induction, $T(Q) \xrightarrow{\mu} T(P')$.

By lemma 3.5, $T(Q) \equiv_a T(P)$.
Hence, by ALP $T(P) \xrightarrow{\mu} T(P')$ as desired.

(ii) Assume $T(P) \xrightarrow{\mu} Q$ by ALP, $Q' \xrightarrow{\mu} Q$, $Q' \equiv_a T(P)$.
Then, by induction $P \xrightarrow{\mu} P'$.

**• OUT:**
(i) Assume $P \xrightarrow{\mu} P'$ by OUT.
Then by OUT $P = \mu P'$ and $T(P) = T(\mu P') = \mu T(P') = \mu T(P') \xrightarrow{\mu} T(P')$.

(ii) Assume $T(P) \xrightarrow{\mu} Q$ by OUT.
Then, necessarily $P = \mu P'$, $Q = T(P')$, and by OUT $P \xrightarrow{\mu} P'$. 

227
• **INP:**
  (i) Assume \( P \xrightarrow{\mu} P' \) by INP.
  Then \( \mu = x(z) \) for some \( x, z \), and \( P = x(z').Q \)
  for some name \( z' \) and agent \( Q \in \Pi B \) such that \( Q\{z'/z'\} = P' \)
  Hence, by INP
  \[ \mathcal{T}(P) = \mathcal{T}(x(z').Q) = x(z').\mathcal{T}(Q) \xrightarrow{\mu} \mathcal{T}(Q)\{z/z'\} = \]
  \[ \mathcal{T}(Q)\{z/z'\}\] (lemma 2.9) \( \equiv \mathcal{T}(P') \).
  (ii) Assume \( \mathcal{T}(P) \xrightarrow{\mu} Q \) by INP.
  Then \( \mu = x(z) \) for some \( x, z \), and necessarily \( P = x(z').Q' \)
  for some \( z, Q' \in \Pi B \) such that \( \mathcal{T}(Q')\{z/z'\} = \mathcal{T}(Q'\{z/z'\}) = Q \).
  Hence by INP \( P = x(z').Q' \xrightarrow{x(z)} Q'\{z/z'\} \equiv P' \).

• **SUM:**
  (i) Assume \( P \xrightarrow{\mu} P' \) by SUM.
  Then \( P = P_1 + P_2 \) for some agents \( P_1 \in \Pi B, P_2 \in \Pi B \), and
  \( P_1 \xrightarrow{\mu} P' \) by a shorter inference.
  By induction \( \mathcal{T}(P_1) \xrightarrow{\mu} \mathcal{T}(P') \).
  Hence, by SUM, \( \mathcal{T}(P) \equiv \mathcal{T}(P_1 + P_2) = \mathcal{T}(P_1) + \mathcal{T}(P_2) \xrightarrow{\mu} \mathcal{T}(P') \).
  (ii) Assume \( \mathcal{T}(P) \xrightarrow{\mu} Q \) by SUM.
  Then necessarily \( P = P_1 + P_2 \) for some agents \( P_1 \in \Pi B, P_2 \in \Pi B \) and
  \( \mathcal{T}(P_1) \xrightarrow{\mu} Q \) by a shorter inference.
  By induction \( P_1 \xrightarrow{\mu} P' \), where \( \mathcal{T}(P') = Q \).
  Hence, by SUM \( P \equiv P_1 + P_2 \xrightarrow{\mu} P' \).

• **PAR, MATCH, REP, RES and BLOCK:** similar

• **OPEN:**
  (i) Assume \( P \xrightarrow{\mu} P' \) by OPEN.
  Then \( \mu = (z)\exists z \) for some names \( x, z \).
  \( P = (z)Q \) for some \( Q \in \Pi B \), and \( Q \xrightarrow{\exists z} P' \) by a shorter inference.
  By induction \( \mathcal{T}(Q) \xrightarrow{\exists z} \mathcal{T}(P') \).
  Hence, by OPEN \( \mathcal{T}(P) = (z)\mathcal{T}(Q) \xrightarrow{\mu} \mathcal{T}(P') \).
  (ii) Assume \( \mathcal{T}(P) \xrightarrow{\mu} Q \) by OPEN.
  Then \( \mu = (z)\exists z \) for some names \( x, z \),
  necessarily \( P = (z)Q' \) for some agent \( Q' \) and
  \( \mathcal{T}(Q') \xrightarrow{\exists z} Q \) by a shorter inference.
  By induction \( Q' \xrightarrow{\exists z} P' \) for some \( P' \in \Pi B \) such that \( \mathcal{T}(P') = Q \).
  Hence, by OPEN \( P = (z)Q' \xrightarrow{\mu} P' \).

**Lemma 3.7:** If \( x \) is a higher-order channel in \( \Pi B \), and \( P \) any closed agent \( \in \Pi B \), then
(i) If \( P \xrightarrow{(y)F} P' \) then \( \mathcal{T}(P) \xrightarrow{(w)w} (\bar{y})(\mathcal{T}(P') | \text{spawn}_w(F)), w \) fresh;
(ii) If \( P \xrightarrow{x(F)} P' \) then \( \mathcal{T}(P) \xrightarrow{x}\mathcal{T}(P') \), for some \( P' \) such that \( P'\{F/W\} = P' \);
(iii) If \( \mathcal{T}(P) \xrightarrow{(w)w} Q \) then \( P \xrightarrow{(y)F} P', \) for some \( P' \) such that \( (\bar{y})(\mathcal{T}(P') | \text{spawn}_w(F)) \sim \)
\( Q \) for some name vector \( \bar{y} \) and abstraction \( F \in \Pi B \);
(iv) if \( T(P) \xrightarrow{\pi(w)} Q \) then \( P \xrightarrow{\pi(F)} P' \{ F/W \} \)
for some agent \( P' \) and any abstraction \( F \in \Pi B \) such that \( T(P') = Q \).

**Proof:** by induction on length of inference.

- **ALP:**
  
  (i) Assume \( P \xrightarrow{(y)\pi(F)} P' \) by ALP, and
  
  \[ Q \xrightarrow{(y)\pi(F)} P', \quad Q \equiv_a P. \]
  
  Then, by induction
  
  \[ T(Q) \xrightarrow{(w)\pi(u)} (\bar{y})(T(P') \mid \text{spawn}_w(F)). \]
  
  Since by lemma 3.5 \( T(Q) \equiv_a T(P) \), by ALP
  
  \[ T(P) \xrightarrow{(w)\pi(u)} (\bar{y})(T(P') \mid \text{spawn}_w(F)) \] as desired.

  (ii) Assume \( P \xrightarrow{\pi(F)} P' \) by ALP, and
  
  \[ Q \xrightarrow{\pi(F)} P', \quad Q \equiv_a P. \]
  
  Then, by induction
  
  \[ T(Q) \xrightarrow{\pi(u)} T(P'). \]
  
  Then necessarily
  
  \[ T(P) \xrightarrow{\pi(u)} T(P') \] as desired.

  (iii) Assume \( T(P) \xrightarrow{(w)\pi(u)} Q \) by ALP.
  
  Then, for some agent \( Q' \in \Pi B \) such that \( T(P) \equiv_a Q' \),
  
  \[ Q' \xrightarrow{(w)\pi(u)} Q. \]
  
  By induction \( Q' \sim (\bar{y})(T(P') \mid \text{spawn}_w(F)) \) for some \( \bar{y} \) and some agents \( F \) and \( P' \)
  
  such that \( P \xrightarrow{(y)\pi(F)} P' \) as desired.

  (iv) Assume \( T(P) \xrightarrow{\pi(w)} Q \) by ALP.
  
  Then, for some agent \( Q' \in \Pi B \) such that \( T(P) \equiv_a Q' \),
  
  \[ Q' \xrightarrow{\pi(w)} Q. \]
  
  By induction \( P \xrightarrow{\pi(F)} P' \{ F/W \} \) for some \( P' \in \Pi B \) such that \( T(P') = Q \).

- **OUT:** Only case (i) is possible, with void \( \bar{y} \).
  
  (i) Assume \( P \xrightarrow{\pi(F)} P' \) by OUT.
  
  Then \( P = \pi(F), P', F \) not a variable since \( P \) is closed.
  
  Thus, by OPEN
  
  \[ T(P) = T(\pi(F), P') = (w)\pi(u). (T(P') \mid \text{spawn}_w(F)) \xrightarrow{(w)\pi(u)} T(P') \mid \text{spawn}_w(F) = (\bar{y})(T(P') \mid \text{spawn}_w(F)). \]

- **INP:** Only (ii) and (iv) possible.
  
  (ii) Assume \( P \xrightarrow{\pi(F)} P' \) by INP.
  
  Then \( P = x(W).P'' \) where \( P'' \{ F/W \} = P' \).
  
  Hence \( T(P) = T(x(W).P'') = x(w). T(P'') \xrightarrow{x(w)} T(P'). \)

  (iv) Suppose \( T(P) \xrightarrow{\pi(w)} Q \) by INP.
  
  Then necessarily \( P \equiv x(Y).P' \) for some \( P' \) such that
  
  \( Q = T(P') \{ w/y \} = T(P' \{ W/Y \}). \)
By IND $P \equiv x(Y), P'^* x^{(F)} P^* \{F/Y\} \equiv P'^* \{W/Y\} \{F/W\}.$

- SUM:

(i) Assume $P \xrightarrow{\{\bar{y}\} x^{(F)}} P'$ by SUM.

Then $P = P_1 + P_2$ for some agents $P_1, P_2 \in \Pi B$, and $P_1 \xrightarrow{\{\bar{y}\} x^{(F)}} P'$.

By induction $T(P_1) \xrightarrow{\{w\} x^{(w)}} (\bar{y})(T(P_1) \mid \text{spawn}_w(F)).$

Hence, by SUM

$$T(P) = T(P_1 + P_2) = T(P_1) + T(P_2) \xrightarrow{\{w\} x^{(w)}} (\bar{y})(T(P_1) \mid \text{spawn}_w(F)).$$

(ii) Assume $P \xrightarrow{\{\bar{y}\} x^{(F)}} P'$ by SUM.

Then for some agents $P_1, P_2 \in \Pi B, P \equiv P_1 + P_2$ and $P_1 \xrightarrow{\{\bar{y}\} x^{(F)}} P'$.

By induction $T(P_1) \xrightarrow{\{w\} x^{(w)}} T(P')$ where $P' = P'^* \{F/W\}$.

Hence, by SUM

$$T(P) = T(P_1 + P_2) = T(P_1) + T(P_2) \xrightarrow{\{w\} x^{(w)}} T(P'),$$

as desired.

(iii) Assume $T(P) \xrightarrow{\{w\} x^{(w)}} Q$ by SUM.

Then necessarily $P = P_1 + P_2$ for some agents $P_1, P_2 \in \Pi B,$ and

$$T(P_1) \xrightarrow{\{w\} x^{(w)}} Q.$$ 

By induction $Q \sim (\bar{y})(T(P') \mid \text{spawn}_w(F))$

for some names $\bar{y}$ and agents $P', F \in \Pi B$ such that $P_1 \xrightarrow{\{\bar{y}\} x^{(F)}} P'$.

Hence by SUM $P = P_1 + P_2 \xrightarrow{\{\bar{y}\} x^{(F)}} P'$ as desired.

(iv) Assume $T(P) \xrightarrow{\{w\} x^{(w)}} Q$ by SUM.

Then necessarily $P = P_1 + P_2$ for some agents $P_1, P_2 \in \Pi B,$ and $T(P_1) \xrightarrow{\{w\} x^{(w)}} Q$.

By induction $P_1 \xrightarrow{\{\bar{y}\} x^{(F)}} P'^* \{F/W\}$ for some agents $F, P' \in \Pi B$ such that $Q = T(P')$.

Hence, by SUM $P \xrightarrow{\{\bar{y}\} x^{(F)}} P'^* \{F/W\}$.

- OPEN: Only cases (i) and (iii) are possible.

(i) Assume $P \xrightarrow{\{\bar{y}\} x^{(F)}} P'$ by OPEN, $\bar{y} = x'\bar{y}'.

Then $P \equiv (x')P'^*$ for some agent $P'^* \in \Pi B$ such that $P'^* \xrightarrow{\{\bar{y}'\} x^{(F)}} P'$.

By induction $T(P'^*) \xrightarrow{\{w\} x^{(w)}} (\bar{y}')(T(P') \mid \text{spawn}_w(F)).$

Hence, by RES

$$T(P) = T((x')P'^*) = (x')T(P'^*) \xrightarrow{\{w\} x^{(w)}} (x')(\bar{y}')(T(P') \mid \text{spawn}_w(F)) = (\bar{y})(T(P) \mid \text{spawn}_w(F)).$$

(ii) Assume $T(P) \xrightarrow{\{w\} x^{(w)}} Q$ by OPEN.

Then necessarily $P = \pi(F), P'^* \{F/W\}$ for some $F, P' \in \Pi B,$ and

$$Q \sim T(P') \mid \text{spawn}_w(F).$$

Hence $P = \pi(F), P'^* \xrightarrow{\{\bar{y}\} x^{(F)}} P'$, where $\bar{y}$ is void and thus $Q \sim (\bar{y})(T(P') \mid \text{spawn}_w(F))$ as desired.

- RES:

(i) Assume $P \xrightarrow{\{\bar{y}\} x^{(F)}} P'$ by RES.
Then, for some name $z \not\in \text{fn}(F)$, $z \neq x$, and some agents $Q$, $Q'$, $P = (z)Q$, $P' = (z)Q'$, and $Q \xrightarrow{(y)\exists w} Q$ by a shorter inference.

Thus, by induction $T(Q) \xrightarrow{(w)\exists w} (\bar{y})(T(Q') \mid \text{spawn}_w(F))$.

Hence, by RES

$$T(P) = T((z)Q) = (z)T(Q) \xrightarrow{(w)\exists w} (z)(\bar{y})(T(Q') \mid \text{spawn}_w(F)) = (\bar{y})(T((z)Q') \mid \text{spawn}_w(F)) = (\bar{y})(T(Q') \mid \text{spawn}_w(F))$$

(since $z \not\in \text{fn}(F)$ implies $z \not\in \text{fn}(\text{spawn}_w(F))$)

$$= (\bar{y})(T((z)Q') \mid \text{spawn}_w(F)) = (\bar{y})(T(P) \mid \text{spawn}_w(F))$$

as desired.

(ii) Assume $P \xrightarrow{x(F)} P'$ by RES.

Then, for some name $z \neq x$ and for some agents $F$, $Q$, $Q' \in \Pi B$, $P = (z)Q$, $P' = (z)Q'$, and $Q \xrightarrow{z(F)} Q'$.

Thus, by induction $T(Q) \xrightarrow{(w)\exists w} T(Q'')$, $Q' \{F/W\} = Q'$.

Hence by RES

$$T(P) = T((z)Q) = (z)T(Q) \xrightarrow{(w)\exists w} (z)T(Q'') = T((z)Q''),$$

and

$$(z)(Q') \{F/W\} = (z)(Q' \{F/W\}) = (z)Q' = P'$$
as desired.

(iii) Assume $T(P) \xrightarrow{(w)\exists w} Q$ by RES.

Then, necessarily for some name $z \neq w$ and some agents $P''$, $Q' \in \Pi B$, $P = (z)P''$, $Q = (z)Q'$, and

$$T(P'') \xrightarrow{(w)\exists w} Q'$$

by a shorter inference.

Thus, by induction, for some $\bar{y}$, $F$ and $Q''$

$$P'' \xrightarrow{(\bar{y})\exists w} Q''$$

where $(\bar{y})(T(Q'') \mid \text{spawn}_w(F)) \sim Q'$.

Assume $z \not\in \text{fn}(F)$. Then, by RES

$$P = (z)P'' \xrightarrow{(\bar{y})\exists w} (z)Q''$$

and

$$(\bar{y})(T((z)Q'') \mid \text{spawn}_w(F)) = (\bar{y})((z)T(Q'') \mid \text{spawn}_w(F)) = (\bar{y})((z)(\bar{y})(T(Q') \mid \text{spawn}_w(F)) (since z \not\in \text{fn}(F) \implies z \not\in \text{fn}(\text{spawn}_w(F))) = (z)(\bar{y})(T(Q') \mid \text{spawn}_w(F)) \sim (z)Q = Q.$$

Assume now that $z \in \text{fn}(F)$. Then, by OPEN

$$P = (z)P'' \xrightarrow{(z,\bar{y})\exists w} Q''$$

and

$$(z,\bar{y})(T(Q'') \mid \text{spawn}_w(F)) = (z)(\bar{y})(T(Q'') \mid \text{spawn}_w(F)) \sim (z)Q' = Q$$
as desired.

(iv) Assume $T(P) \xrightarrow{x(F)} Q$ by RES.

Then, necessarily for some name $z \neq w$, and some agents $P''$, $Q' \in \Pi B$, $P = (z)P''$, $Q = (z)Q'$, and

$$T(P'') \xrightarrow{x(F)} Q'$$

by a shorter inference.

Thus, by induction

$$P'' \xrightarrow{x(F)} Q' \{F/W\}$$

where $T(Q'') = Q'$.

We may assume that $z \not\in \text{fn}(F)$. Hence, by RES

$$P = (z)P'' \xrightarrow{x(F)} (z)(Q'' \{F/W\}) = ((z)Q'') \{F/W\}$$

and $T((z)Q'') = (z)T(Q'') = (z)Q' = Q$ as desired.

• PAR:
(i) Assume \( P \xrightarrow{(y)\mathcal{P}_F} P' \) by PAR. Then necessarily
\[
P = P_1 | P_2 \text{ for some agents } P_1, P_2 \in \Pi B, \ y \cap \text{fn}(P) = \emptyset; \]
\[
P' = P'_1 | P_2 \text{ for some agent } P'_1 \in \Pi B \text{ such that} \]
\[
P_1 \xrightarrow{(y)\mathcal{P}_F} P'_1 \text{ by a shorter inference.} \]

By induction \( T(P_1) \xrightarrow{(w)\mathcal{P}_w} (y)(T(P'_1) | \text{spawn}_w(F)). \)
Then, by PAR
\[
T(P) = T(P_1 | P_2) = T(P_2) \xrightarrow{(w)\mathcal{P}_w} (y)(T(P'_1) | \text{spawn}_w(F)) | T(P_2) = (y)(T(P'_1) | T(P_2) | \text{spawn}_w(F)) \]
(since \( y \cap \text{fn}(P_2) = \emptyset \Rightarrow y \cap \text{fn}(T(P_2)) = \emptyset \))
\[
(y)(T(P'_1) | \text{spawn}_w(F)) = (y)(T(P') | \text{spawn}_w(F)). \]

(ii) Assume \( P \xrightarrow{x(F)} P' \) by PAR.
Then, for some agents \( P_1, P_2, P'_1 \in \Pi B, \ P = P_1 | P_2, \ P' = P'_1 | P_2 \) and
\[
P_1 \xrightarrow{x(F)} P'_1 \text{ by a shorter inference.} \]
Thus, by induction
\[
T(P_1) \xrightarrow{x(w)} T(P') \text{ where } P' \{F/W\} = P'_1. \]
Hence, by PAR, since \( W \) does not occur free in \( P_2 \) because it is closed,
\[
T(P) = T(P_1 | P_2) = T(P_1) \xrightarrow{x(w)} T(P') | T(P_2) = T(P') | P_2 \]
and
\[
(P' \{F/W\} | P_2) = P'' \{F/W\} | P'_1 | P_2 = P' \text{ as desired.} \]

(iii) Assume \( T(P) \xrightarrow{(w)\mathcal{P}_w} Q \) by PAR.
Then for some agents \( P_1, P_2 \in \Pi B, \ Q' \in \pi B, \)
\[
P = P_1 | P_2, \ w \notin \text{fn}(T(P_2)), \ Q = Q' | T(P_2) \text{ and} \]
\[
T(P_1) \xrightarrow{(w)\mathcal{P}_w} Q' \text{ by a shorter inference.} \]
Thus, by induction
\[
P_1 \xrightarrow{(y)\mathcal{P}_F} P'_1 \text{ where } (y)(T(P'_1) | \text{spawn}_w(F)) \sim Q'. \]
Thus by PAR, assuming \( y \cap \text{fn}(P_2) = \emptyset, \)
\[
P_1 \xrightarrow{(y)\mathcal{P}_F} P'_1 \ | P_2, \]
and
\[
(y)(T(P'_1) | \text{spawn}_w(F)) = (y)(T(P'_1) | T(P_2) | \text{spawn}_w(F)) = (y)(T(P'_1) | \text{spawn}_w(F)) | T(P_2) \]
(since \( y \cap \text{fn}(P_2) = \emptyset \Rightarrow y \cap \text{fn}(T(P_2)) = \emptyset \))
\[
\sim Q' | T(P_2) = Q. \]

(iv) Assume \( T(P) \xrightarrow{x(w)} Q \) by PAR.
Then necessarily for some agents \( P_1, P_2 \in \Pi B, P'_1 \in \pi B \)
\[
P = P_1 | P_2, \ Q = P'_1 | T(P_2), \text{ and } T(P_1) \xrightarrow{x(w)} P'_1 \]
by a shorter inference. Thus, by induction \( P_1 \xrightarrow{x(F)} P_1'' \{F/W\} \)
for some agents \( P''_1, F \in \Pi B \) such that \( T(P''_1) = P'_1 \).
Hence, by PAR, since \( W \) does not occur free in the closed agent \( P_2, \)
\[
P = P_1 | P_2 \xrightarrow{x(F)} P''_1 \{F/W\} | P_2 = (P''_1 | P_2) \{F/W\} \text{ and} \]
\[
T(P''_1) | P_2 = T(P''_1) | T(P_2) = P'_1 | T(P_2) = Q. \]

\textbf{MATCH:} \\
(i) Assume \( P \xrightarrow{(y)\mathcal{P}_F} P' \) by MATCH.
Then for some name $z$ and some agent $P' \in \Pi B$, $P = [z = z]P''$ and
$P'' \xrightarrow{[w]w} P'$ by a shorter inference.
Thus, by induction $T(P'') \xrightarrow{[w]w} (\bar{y})(T(P') \mid spawn_w(F))$.
Hence, by MATCH
$T(P) = T([z = z]P'') = [z = z]T(P'') \xrightarrow{[w]w} (\bar{y})(T(P') \mid spawn_w(F))$.

(ii) Assume $P \xrightarrow{x(F)} P'$ by MATCH.
Then for some name $z$ and some agent $P' \in \Pi B$, $P = [z = z]P''$ and
$P'' \xrightarrow{x(F)} P'$ by a shorter inference.
Thus, by induction $T(P'') \xrightarrow{x(F)} T(Q)$ where $Q\{F/W\} = P'$.
Hence, by MATCH
$T(P) = T([z = z]P'') = [z = z]T(P'') \xrightarrow{x(F)} T(Q)$.

(iii) Assume $T(P) \xrightarrow{[w]w} Q$ by MATCH.
Then for some name $z$ and some agent $P' \in \Pi B$, $P = [z = z]P''$ and
$T(P'') \xrightarrow{[w]w} Q$ by a shorter inference.
Thus, by induction
$P'' \xrightarrow{[w]w} P', \bar{y}(T(P') \mid spawn_w(F)) \sim Q$.
Hence, by MATCH $P = [z = z]P'' \xrightarrow{[w]w} P'$.

(iv) Assume $T(P) \xrightarrow{x(F)} Q$ by MATCH.
Then for some name $z$ and some agent $P' \in \Pi B$, $P = [z = z]P''$ and
$T(P'') \xrightarrow{x(F)} P' \{F/W\}$, where $T(P') = Q$.
Hence, by MATCH $P = [z = z]P'' \xrightarrow{x(F)} P' \{F/W\}$.

• REP:

(i) Assume $P \xrightarrow{[y]w(F)} P'$ by REP.
Then for some agent $P'' \in \Pi B$, $P =!P''$ and
$P'' \xrightarrow{[y]w(F)} P'$ by a shorter inference.
Thus, by induction
$T(P'') \xrightarrow{[y]w(F)} !T(P'') \xrightarrow{[y]w(F)} (\bar{y})(T(P') \mid spawn_w(F))$.
Hence, by REP
$T(P) = T(P'') =!T(P'') \xrightarrow{[y]w(F)} (\bar{y})(T(P') \mid spawn_w(F))$.

(ii) Assume $P \xrightarrow{x(F)} P'$ by MATCH.
Then for some agent $P'' \in \Pi B$, $P =!P''$ and
$P'' \xrightarrow{x(F)} P'$ by a shorter inference.
Thus, by induction $T(P'') \xrightarrow{x(F)} T(P'') \xrightarrow{x(F)} T(Q)$
where $Q\{F/W\} = P'$.
Hence, by REP
$T(P) = T(P'') =!T(P'') \xrightarrow{x(F)} T(Q)$.

(iii) Assume $T(P) \xrightarrow{[w]w} Q$ by REP.
Then for some agent $P'' \in \Pi B$, $P =!P''$ and
$T(P'') \xrightarrow{[w]w} T(P'') \xrightarrow{[w]w} Q$ by a shorter inference.
Thus, by induction
\[ P' \vdash P' \overset{(\overline{y}\overline{F})}{\rightarrow} \tilde{P}, (\overline{y})(T(P')) \mid spawn_w(F)) \sim Q. \]

Hence, by REP \( P = P' \overset{(\overline{y}\overline{F})}{\rightarrow} P' \).

(iv) Assume \( T(P) \overset{\alpha}{\rightarrow} Q \) by REP.

Then for some agent \( P'' \in \Pi B \), \( P' = P'' \) and

\[ \mathcal{T}(P'' \upharpoonright P') = \mathcal{T}(P'') \mid \mathcal{T}(P'') \overset{\alpha}{\rightarrow} Q \text{ by a shorter inference}. \]

Thus, by induction \( P'' \vdash P'' \overset{\alpha}{\rightarrow} P' \{F/W\} \), where \( T(P') = Q \).

Hence, by REP \( P = P'' \overset{\alpha}{\rightarrow} P' \{F/W\} \).

**BLOCK:**

(i) Assume \( P \overset{(\overline{y}\overline{F})}{\rightarrow} P' \) by BLOCK.

Then for some name \( z \neq x \) and some agents \( Q, Q' \in \Pi B \),

\[ P = Q \backslash z, P' = Q' \backslash z \text{ and } Q \overset{(\overline{y}\overline{F})}{\rightarrow} Q', \]

by a shorter inference.

Thus, by induction \( \mathcal{T}(Q) \overset{\alpha}{\rightarrow} (\overline{y})(T(Q') \mid spawn_w(F)) \).

Hence, by BLOCK

\[ \mathcal{T}(Q) \overset{\alpha}{\rightarrow} (\overline{y})(T(Q') \mid spawn_w(F)) \mid z =
\]

\[ (\overline{y})(T(Q') \mid spawn_w(F)) \mid z \text{ (since we may assume } z \notin \overline{y}). \]

Since the possible subjects for actions of \( spawn_w(F) \) are either \( w \) or other spawning and sender channels, \( z \) is never the subject of any of these actions. Hence

\[ (\overline{y})(T(Q') \mid spawn_w(F)) \mid z \approx (\overline{y})(T(Q') \mid spawn_w(F)) =
\]

\[ (\overline{y})(T(Q') \mid spawn_w(F)) = (\overline{y})(T(P') \mid spawn_w(F)) \]

as desired.

(ii) Assume \( P \overset{\alpha}{\rightarrow} P' \) by BLOCK.

Then for some name \( z \neq x \) and some agents \( Q, Q' \in \Pi B \), \( P = Q \backslash z, P' = Q' \backslash z \), and

\[ Q \overset{\alpha}{\rightarrow} Q' \text{ by a shorter inference}. \]

Thus, by induction

\[ \mathcal{T}(Q) \overset{\alpha}{\rightarrow} T(Q') \text{ where } Q' \{F/W\} = Q'. \]

Hence, by BLOCK

\[ \mathcal{T}(P) = \mathcal{T}(Q) \backslash z = \mathcal{T}(Q) \overset{\alpha}{\rightarrow} T(Q') \backslash z = T(Q') \backslash z\]

and \( (Q' \backslash z) \{F/W\} = Q' \{F/W\} \backslash z = Q' \backslash z = P' \) as desired.

(iii) Assume \( T(P) \overset{\alpha}{\rightarrow} Q \) by BLOCK.

Then for some name \( z \neq x \), and some agents \( P'', Q' \in \Pi B \),

\[ P = P'' \backslash z, Q = Q' \backslash z \]

and

\[ T(P') \overset{\alpha}{\rightarrow} Q' \text{ by a shorter inference}. \]

Thus, by induction

\[ P'' \overset{\overline{y}\overline{F}}{\rightarrow} Q'', (\overline{y})(T(Q'') \mid spawn_w(F)) \sim Q'. \]

We may assume \( z \notin \overline{y}. \)

Hence, by BLOCK \( P = P'' \backslash z \overset{\overline{y}\overline{F}}{\rightarrow} Q'' \backslash z \) and

\[ (\overline{y})(T(Q'') \backslash z \mid spawn_w(F)) = (\overline{y})(T(Q'') \backslash z \mid spawn_w(F)) =
\]

\[ (\overline{y})(T(Q'') \mid spawn_w(F)) \mid z \text{ (as above in (i))} \approx
\]

\[ ((\overline{y})(T(Q')) \mid spawn_w(F)) \mid z \text{ (since } z \notin \overline{y}) \sim
\]

\[ Q' \backslash z = P' \] as desired.

234
(iv) Assume \( T(P) \xrightarrow{\pi} Q \) by BLOCK.
Then for some name \( z \neq x \) and some agents \( P' \), \( Q' \in \Pi B \),
\[ P = P'{\backslash}_{z}, \quad Q = Q'{\backslash}_{z}, \text{ and} \]
\[ T(P') \xrightarrow{\pi} Q' \] by a shorter inference.
Thus, by induction \( P' \xrightarrow{\pi} Q' \{F/W\} \), where \( T(Q') = Q' \).
Hence, by BLOCK
\[ P = P'{\backslash}_{z} \xrightarrow{\pi} (Q' \{F/W\}){\backslash}_{z} = (Q'{\backslash}_{z})\{F/W\} \]
and \( T(Q'{\backslash}_{z}) = T(Q'){\backslash}_{z} = Q' \) as desired.
The following definitions follow closely [113]. We use \( P \xrightarrow{\Delta} P' \) to mean \( P \rightarrow P' \) or \( P \equiv P' \), and \( \Rightarrow \) to mean the transitive closure of \( \rightarrow \).

**Definition 3.5 (Expansion):** \( \varepsilon \) is an expansion if \( \forall \varepsilon \) implies:
1. Whenever \( P \rightarrow P' \), then \( Q' \) exists s.t. \( Q \Rightarrow P' \varepsilon Q' \), and for each \( a \), if \( P' \downarrow a \) then \( Q' \downarrow a \);
2. Whenever \( Q \rightarrow Q' \), then \( P' \) exists s.t. \( P \xrightarrow{\Delta} P' \) and \( P' \varepsilon Q' \), and for each \( a \), if \( Q' \downarrow a \) then \( P' \downarrow a \);

We say that \( Q \) expands \( P \), written \( P \preceq Q \), if \( \exists \varepsilon \) for some expansion \( \varepsilon \).
We define also \( P \preceq Q \) if for all reduced contexts \( C[\_] \) we have \( C[P] \preceq C[Q] \).

**Definition 3.6 (Weak Barbed Bisimulation up to Expansion):** \( S \) is a weak barbed bisimulation up to expansion if whenever \( P \vDash S \) then:
1. Whenever \( P \rightarrow P' \), then \( Q' \) exists s.t. \( Q \Rightarrow P' \vDash P' \approx Q' \).
2. Whenever \( Q \rightarrow Q' \), then \( P' \) exists s.t. \( P \rightarrow P' \approx Q' \).

Note that \( \sim \subseteq \sim_{r} \subseteq \approx \).

In the proof of following three lemmas, 3.8, 3.9 and 3.10, all equivalences used are preserved by reduced contexts. We note also that \( \leq \) is preserved by all operators except input prefixing and unguarded sums, and also by input prefixing in case it is preserved by arbitrary substitutions of the input variable.

**Lemma 3.8:** \( \text{send}_{a}(P{\{F/W\}}) \leq (w)(\text{spawn}_{a}(F) \mid \text{send}_{a}(P)) \).

**Corollary 1:** For any higher-order abstraction \( G \) which is not a process variable, \( \text{spawn}_{a}((G \{F/W\})) \approx_{r} (w)(\text{spawn}_{a}(F) \mid \text{spawn}_{a}(G)) \).

**Proof:** Let \( G = (\lambda X)P \) for some process variable \( X \neq W \) and process \( P \in \Pi B \). Then
\[ \text{spawn}_{a}(G \{F/W\}) = \text{spawn}_{a}((\lambda X)P \{F/W\}) = \]
\[ !u'(u).u.(v).u.(x).u.(v).u.(x).u.(w)(\text{spawn}_{a}(F) \mid \text{send}_{a}(P)) \]
by lemma and since thus \( \text{send}_{a}(P{\{F/W\}}) \{v'/v\} = \text{send}_{a}(P \{F/W\}) \leq (w)(\text{spawn}_{a}(F) \mid \text{send}_{a}(P)) \{v'/v\} \)
and \( \text{send}_{a}(P{\{F/W\}}) \{x'/x\} \leq (w)(\text{spawn}_{a}(F) \mid \text{send}_{a}(P)) \{x'/x\} \)
since this implies only a renaming of agent variable \( X \) to \( X' \) in \( \lambda X.P \).
Further, \( !u'(u).u.(v).u.(x).u.(w)(\text{spawn}_{a}(F) \mid \text{send}_{a}(P)) \]
\[ \sim_{r} (w)(\text{spawn}_{a}(F) \mid !u'(u).u.(v).u.(x).u.(w)(\text{spawn}_{a}(P)) \text{ (by expansion)} \]
\[ \sim_{r} (w)(\text{spawn}_{a}(F) \mid !u'(u).u.(v).u.(x).u.(w)(\text{spawn}_{a}(P)) \text{ (lemma 5(iii))} \]
\[ = (w)(\text{spawn}_{a}(F) \mid \text{spawn}_{a}((\lambda X)P)) = (w)(\text{spawn}_{a}(F) \mid \text{spawn}_{a}(G)). \]
Corollary 2: \((w)\text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel \text{spawn}_w(F)\parallel \text{spawn}_w(G)\parallel \leq (w)\text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel\) for any reduced composition \(P\).

Proof: For any reduced composition \(P\), assuming \(w\) is fresh,
\[
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel \sim \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel \parallel = \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel \parallel \leq \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel (w)\text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel \parallel \sim \parallel \text{by cor. 1} \parallel \sim \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel \parallel \parallel \text{by lemma 3.3(i)} \parallel \sim \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{spawn}_w(G)\parallel \parallel \text{as desired}.
\]

Proof of lemma:

Structural induction on \(P\)’s formation.

\(\bullet P = 0.\) Then
\[
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P)\parallel = (w)\text{spawn}_w(F) \parallel \text{send}_w(0)\parallel = \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P)\parallel = \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(0)\parallel \parallel \text{by expansion} \parallel \sim \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P)\parallel \parallel \text{as desired}.
\]

\(\bullet P = \text{pp}.\) Then
\[
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P)\parallel = (w)\text{spawn}_w(F) \parallel \text{send}_w(\text{pp})\parallel = \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(\text{pp})\parallel \parallel \text{by expansion} \parallel \sim \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P)\parallel \parallel \text{as desired}.
\]

\(\bullet P = \text{pp}.\) Then
\[
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P)\parallel = (w)\text{spawn}_w(F) \parallel \text{send}_w(a(b).P')\parallel = \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(a(b).P')\parallel \parallel \text{by expansion} \parallel \sim \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P')\parallel \parallel \text{as desired}.
\]

\(\bullet P = \text{pp}\). Similar.

\(\bullet P = \text{pp}.\) Then
\[
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P)\parallel = (w)\text{spawn}_w(F) \parallel \text{send}_w(\text{pp}\{W\}).P'\parallel = \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(\text{pp}\{W\}).P'\parallel \parallel \text{by expansion} \parallel \sim \\
(\gamma')\parallel \gamma \parallel \text{spawn}_w(F) \parallel \text{send}_w(P')\parallel \parallel \text{as desired}.
\]

\(\bullet P = a(\text{pp}).\) Similar.
\[ \forall o. Pa.(w). (send_o(P \{ F / W \} \mid spawn_o(F)) \leq \]
\[ \forall o. Pa.(w). ((w) \mid spawn_o(F) \mid send_o(P) \mid spawn_o(F)) \ (by \ induction) \sim_r \]
\[ \forall o. Pa.(w). (send_o(P') \mid spawn_o(F)) \ (by \ lemma \ 3.3(i)). \]

Assume now that \( Y \neq W \). Then
\[ (w) (\text{spawn}_o(F) \mid send_o(P)) = (w)(\text{spawn}_o(F) \mid send_o(\pi(Y), P')) = \]
\[ (w)(\text{spawn}_o(F) \mid \forall o. Pa.\tau_y. send_o(P')) \sim \]
\[ \forall o. Pa.\tau_y.(w)(\text{spawn}_o(F) \mid send_o(P')) \ (by \ expansion). \]
Also
\[ send_o(P \{ F / W \}) = send_o(\pi(\pi(Y), P') \{ F / W \}) = send_o(\pi(\pi(Y), P' \{ F / W \}) = \]
\[ \forall o. Pa.\tau_y. send_o(P' \{ F / W \}) \leq \]
\[ \forall o. Pa.\tau_y. (w)(\text{spawn}_o(F) \mid send_o(P')). \]

\[ \bullet P = \pi(G), P' \text{ not a process variable. Then} \]
\[ (w)(\text{spawn}_o(F) \mid send_o(P)) = (w)(\text{spawn}_o(F) \mid send_o(\pi(G), P')) = \]
\[ (w)(\text{spawn}_o(F) \mid \forall o. Pa.\tau_u. (send_o(P') \mid \text{spawn}_o(G))) \sim \]
\[ \forall o. Pa.\tau_u.(w)(\text{spawn}_o(F) \mid send_o(P') \mid \text{spawn}_o(G)) \ (by \ expansion) \]
\[ \geq \forall o. Pa.\tau_u.(w)(\text{spawn}_o(F) \mid send_o(P') \mid \text{spawn}_o(G \{ F / W \})) \ (by \ cor. 2) \]
\[ \sim \forall o. Pa.\tau_u.(w)(\text{spawn}_o(F) \mid send_o(P') \mid \text{spawn}_o(G \{ F / W \})) \]
\[ (\text{since } w \not\in \text{fn}(\text{spawn}_o(G \{ F / W \}))) \]
\[ \forall o. Pa.\tau_u.(w)(send_o(P' \{ F / W \}) \mid \text{spawn}_o(G \{ F / W \})) \ (by \ induction) = \]
\[ send_o(\pi(\pi(G \{ F / W \}), P') \{ F / W \}) = send_o(P \{ F / W \}). \]

\[ \bullet P = \sum_{i=1}^n \alpha_i P_i. \text{ Then} \]
\[ (w)(\text{spawn}_o(F) \mid send_o(P)) = (w)(\text{spawn}_o(F) \mid send_o(\sum_{i=1}^n \alpha_i P_i)) = \]
\[ (w)(\text{spawn}_o(F) \mid \forall o. Pa.\tau_{(v_1), \tau_{(v_2)}}(send_o(\sum_{i=1}^n \alpha_i P_i))) \sim \]
\[ \forall o. Pa.\tau_{(v_1), \tau_{(v_2)}}((w)(\text{spawn}_o(F) \mid send_o(P_1)) \mid (w)(\text{spawn}_o(F) \mid send_o(\sum_{i=2}^n \alpha_i P_i))) \]
\[ (by \ lemma \ 3.3(ii)) \geq \]
\[ \forall o. Pa.\tau_{(v_1), \tau_{(v_2)}}(send_o(P_1 \{ F / W \}) \mid send_o(\sum_{i=2}^n \alpha_i P_1 \{ F / W \})). \]
\[ (by \ induction) = \]
\[ send_o(\sum_{i=1}^n \alpha_i P_i \{ F / W \}) = \]
\[ send_o(\sum_{i=1}^n \alpha_i P_i \{ F / W \}) = send_o(P \{ F / W \}) \]
as desired.

\[ \bullet P = P_1 \mid P_2. \text{ Then} \]
\[ (w)(\text{spawn}_o(F) \mid send_o(P)) = (w)(\text{spawn}_o(F) \mid send_o(P_1 \mid P_2)) = \]
\[ (w)(\text{spawn}_o(F) \mid \forall o. Pa.\tau_{(v_1), \tau_{(v_2)}}(send_o(P_1 \mid P_2))) \sim \]
\[ \forall o. Pa.\tau_{(v_1), \tau_{(v_2)}}((w)(\text{spawn}_o(F) \mid send_o(P_1 \mid P_2)) \mid (w)(\text{spawn}_o(F) \mid send_o(P_2))) \]
\[ (by \ lemma \ 3.3(ii)) \geq \forall o. Pa.\tau_{(v_1), \tau_{(v_2)}}(send_o(P_1 \{ F / W \}) \mid send_o(P_2 \{ F / W \})). \]
\[ (by \ induction) = \]
\[ send_o(P_1 \{ F / W \} \mid P_2 \{ F / W \}) = \]
\[ send_o(P \{ F / W \}) = send_o(P \{ F / W \}). \]

\[ \bullet P = !P'. \text{ Then} \]
\[ (w)(\text{spawn}_o(F) \mid send_o(P)) = (w)(\text{spawn}_o(F) \mid send_o(!P')) = \]
\[ (w)(\text{spawn}_o(F) \mid \forall o. Pa.\tau_{(v')}.send_o(P')) \sim \]
\[ \forall o. Pa.\tau_{(v')}.(w)(\text{spawn}_o(F) \mid \pi(v').send_o(P')) \ (by \ expansion) \]
\[ \sim \forall o. Pa.\tau_{(v')}.(w)(\text{spawn}_o(F) \mid \pi(v').send_o(P')) \ (lemma \ 3.3(iii)) \sim \]
\[ \forall o. Pa.\tau_{(v')}.(w)(\text{spawn}_o(F) \mid send_o(P')). \ (by \ expansion) \]

237
\[ \geq \forall r.(\forall d'. send_e(P' \{F/W\}) \text{ (by induction)} = send_e(l(P' \{F/W\})) = send_e(P' \{F/W\}) = send_e(P \{F/W\}). \]

\[ P = (z)P'. \text{ Then } \]
\[ (w)(\text{spawn}_w(F) \mid send_e(P)) = (w)(\text{spawn}_w(F) \mid send_e((z)P')) = (w)(\text{spawn}_w(F) \mid (z)send_e(P')) \sim (z)(w)(\text{spawn}_w(F) \mid send_e(P)) \geq (z)send_e((P \{F/W\}) \text{ (by induction)} = send_e((z)(P' \{F/W\})) = send_e((z)P' \{F/W\}) = send_e(P \{F/W\}). \]

\[ P = [a = b]P'. \text{ Then } \]
\[ (w)(\text{spawn}_w(F) \mid send_e(P)) = (w)(\text{spawn}_w(F) \mid send_e([a = b]P')) = (w)(\text{spawn}_w(F) \mid \forall m. m. \forall z. send_e(P')) \sim \forall m. m. \forall z. (w)(\text{spawn}_w(F) \mid send_e(P')) \text{ (by expansion)} \]
\[ \geq \forall m. m. \forall z. send_e(P' \{F/W\}) \text{ (by induction)} = send_e((P' \{F/W\}) \{z\}) = send_e((P \{F/W\}) \{z\}) = send_e(P \{F/W\}). \]

\[ P = Y(x). \text{ Similar.} \]

\[ P = Y(G), \text{ G not a variable.} \]

Assume \( Y \neq W \). Then
\[ P = P \{F/W\} \text{ and } w \notin \text{fin}(send_e(P)). \text{ Thus } \]
\[ (w)(\text{spawn}_w(F) \mid send_e(P)) \sim (w)\text{spawn}_w(F) \mid send_e(P) \sim 0 \mid send_e(P) \sim send_e(P) = send_e(P \{F/W\}). \]

Assume now that \( Y = W \). Then
\[ (w)(\text{spawn}_w(F) \mid send_e(P)) = (w)(\text{spawn}_w(F) \mid send_w(W(x))) = (w)(\text{spawn}_w(F) \mid w(u). u(v). u(x). send_w(F(x)) \mid \forall w. \forall x. 0) \sim \forall y. y. \forall z. (w)(\text{spawn}_w(F) \mid send_w(F(x))) \text{ (by expansion)} \]
\[ \geq (w)(\text{spawn}_w(F) \mid send_w(F(x))) \sim (w)(\text{spawn}_w(F) \mid send_w(F(x)) \text{ (since } W \text{ does not occur free in } F \text{ if } F \text{ is well-sorted}) \sim send_w(F(x)) = send_w(\{W(x)\} \{F/W\}) = send_e(P \{F/W\}). \]

\[ P = Y(X) \text{. Similar.} \]

\[ P = Y(G), \text{ G not a variable.} \]

Assume \( Y \neq W \). Then
\[ (w)(\text{spawn}_w(F) \mid send_e(P)) = (w)(\text{spawn}_w(F) \mid send_e(Y(G))) = (w)(\text{spawn}_w(F) \mid \forall y. \forall w. \forall (w'). \text{spawn}_w(G)) \sim \forall y. y. \forall w. (w)(\text{spawn}_w(F) \mid \text{spawn}_w(G)) \text{ (by expansion)} \]
\[ \geq \forall y. y. \forall w. (w)(\text{spawn}_w(F) \mid \text{spawn}_w(G \{F/W\})) \text{ (by cor. 2)} \sim \forall y. y. \forall w. (w)(\text{spawn}_w(F) \mid \text{spawn}_w(G \{F/W\}) \text{ (since } w \notin \text{fin}(\text{spawn}_w(G \{F/W\}))) \sim \forall y. y. \forall w. (w)(\text{spawn}_w(G \{F/W\}) = send_e(Y(G \{F/W\})) = \]

\[ 238 \]
send\(_v((Y(G))\{F/W\}) = send\(_v(P\{F/W\})\).

Assume now that \(Y = W, F = (\lambda X)Q, X : G\).

Then \(W\) does not occur in \(G\) if \(P\) is well-sorted. Hence
\[P\{F/W\} = F(G)\text{ and }send\(_v(P\{F/W\}) = send\(_v(F(G))\).

Here we use induction on the order of abstraction of \(F\). So far we have shown that the lemma is true for all agents \(P \in \Pi\) and all abstractions \(F\) such that the \(F\) is a first-order abstraction.

Now
\[
(w)(\text{spawn}_w(F) | send\(_v(P)) = (w)(\text{spawn}_w(F) | send\(_v(W(G))) \sim \\
(w)(\text{spawn}_w(F) | w(u).u(v).x_.send\(_v(Q) | \text{rec}(u).x_.\text{send}(\lambda x)G(x)).
\]
\[
\tau.\tau.(ux)(\text{spawn}_w(F) | send\(_v(Q) | \text{spawn}_w(G)) \text{ (by expansion)}
\]
\[
\geq (x)(send\(_v(Q) | \text{spawn}_w(G)) \geq send\(_v(Q\{G/X\})
\]
(by induction, since the order of \(G\) is less than \(F\) if \(P\) is well-sorted)
\[= send\(_v((\lambda X)Q(G)) = send\(_v(F(G)) = send\(_v(P\{F/W\})\).
\]

**Lemma 3.9:**

(i) \((w)(\text{spawn}_w(F) | T(W(x))) \geq T(F(x)), \text{ where } F = (\lambda y)P.\)

(ii) \((w)(\text{spawn}_w(F) | T(W(G))) \geq T(F(G)), \text{ where } F = (\lambda Y)P.\)

**Proof of (i):** \((w)(\text{spawn}_w(F) | T(W(x))) \sim T(F(x)), \text{ where } F = (\lambda y)P.\)

We prove rather that \((v)(send\(_v(P\{x/y\}) | \text{rec}(v)) \geq T(F(x)), \text{ since then}
\[
(w)(\text{spawn}_w((\lambda y)P) | T(W(x))) = \\
(w)(\text{spawn}_w((\lambda y)P) | w(u).u(v).y_.send\(_v(P) | \text{rec}(u).y_.\text{rec}(v)) \sim \\
\tau.\tau.(uv)(\text{spawn}_w((\lambda y)P) | send\(_v(P\{x/y\}) | \text{rec}(v)) \geq \\
(v)(send\(_v(P\{x/y\}) | \text{rec}(v))
\]
(by lemma 3.4 and since \(w \not\in \text{fn}(send\(_v(P\{x/y\}) | \text{rec}(v))
\]
\[\geq T(F(x)) \text{ as desired.}
\]

The proof is by induction on \(P\)'s formation

- **\(P = 0\). Then**
  \[
  (w)(send\(_v(P\{x/y\}) | \text{rec}(v)) = (v)(send\(_v(0) | (\lambda y)P = \\
  (v)(v) 0 | \text{rec}(v)) \sim \tau.(v)(0 | 0) = 0 = T(0) = T(F(x)).
  \]

- **\(P = \pi_0 P', F = (\lambda y)\pi_0 P'\). Then**
  \[
  (v)(send\(_v(P\{x/y\}) | \text{rec}(v)) = (v)(send\(_v(\pi_0 P'\{x/y\}) | \text{rec}(v))
  \]
  (by lemma 3.4, \(a' = a\{x/y\}, b' = b\{x/y\})
  \[
  = (v)(\pi_0 v.b'.send\(_v(P\{x/y\}) | \text{rec}(v)) \sim \\
  \tau.\tau.(v)(send\(_v(P\{x/y\}) | \pi_0 b'.\text{rec}(v)) \text{ (by expansion)}
  \]
  \[
  \geq (v)(send\(_v(P\{x/y\}) | \pi_0 b'.\text{rec}(v)) \sim \\
  \pi_0 b'.T(P\{x/y\}) \text{ (by induction)}
  \]
  \[
  = T(\pi_0 b'.P\{x/y\}) = T(F(x)).
  \]

- **\(P = a(b).P' F = (\lambda y)a(b).P'\). Then**
  \[
  (v)(send\(_v(P\{x/y\}) | \text{rec}(v)) = (v)(send\(_v(a'(b).P'\{x/y\}) | \text{rec}(v))
  \]
  (by lemma 3.4, where \(a' = a\{x/y\})
  \[
  = (v)(\pi_0 v.b.a'.send\(_v(P'\{x/y\}) | \text{rec}(v)) \sim \\
  \tau.\tau.(v)(v.b.send\(_v(P'\{x/y\}) | a'(b).\pi_0 \text{rec}(v)) \text{ (by expansion)}
  \]
  \[
  \geq a'(b).v.(v.b.send\(_v(P'\{x/y\}) | \pi_0 \text{rec}(v)) \sim \\
  a'(b).v.(v.b.send\(_v(P'\{x/y\}) | \pi_0 \text{rec}(v)) \sim \\
  239
\[a'(b) \tau(v)(send_a(P' \{x/y\}) | rec(v)) \geq \]
\[a'(b)(v)(send_a(P' \{x/y\}) | rec(v)) \geq \]
\[a'(b)T(P' \{x/y\}) \text{ (by induction)} = T(a'(b)P' \{x/y\}) = T(F(x)).\]

\[P = \pi(X'). P', \quad F = (\lambda y)\pi(X'). P'. \quad \text{Then} \]
\[(v)(send_a(P' \{x/y\}) | rec(v)) = (v)(send_a(\pi'(X'). P' \{x/y\}) | rec(v)) \]
(by lemma 3.4, where \(a' = a(x/y)\))
\[= (v)(\pi o \pi.a'. \piuated.a. send_a(P' \{x/y\}) | rec(v)) \]
\[
\sim (v)(\pi o \pi.a'. \piateau.a. send_a(P' \{x/y\}) | rec(v)) \sim \]
\[\tau.\pi.\tau.(v)(send_a(P' \{x/y\}) | a'x'. rec(v)) \sim \]
(by expansion)
\[\geq (v)(send_a(P' \{x/y\}) | a'x'. rec(v)) \sim \]
\[a'x'. T(P' \{x/y\}) \text{ (by induction)} \]
\[= T(\pi'(X')). P' \{x/y\}) = T(F(x)).\]

\[P = \pi(G). P', \quad F = (\lambda y)\pi(G). P', \quad G \text{ not a variable. Then} \]
\[(v)(send_a(P' \{x/y\}) | rec(v)) = (v)(send_a(\pi(G \{x/y\}). P' \{x/y\}) | rec(v)) \]
(by lemma 3.4, where \(a' = a(x/y)\))
\[= (v)(\pi o \pi.a'. \piateau.a'. spawn_w(G \{x/y\}) | rec(v)) \]
\[
\sim \tau.\pi.\tau.(v)(send_a(P' \{x/y\}) | spawn_w(G \{x/y\}) | rec(v)) \sim \]
(by expansion)
\[\geq (v)(send_a(P' \{x/y\}) | spawn_w(G \{x/y\}) | rec(v)) \sim \]
\[a'(w')(v)(send_a(P' \{x/y\}) | spawn_w(G \{x/y\}) | rec(v)) \geq \]
\[a'(w')(v)T(P' \{x/y\}) | spawn_w(G \{x/y\}) \text{ (by induction)} \]
\[
\sim T(a'(G \{x/y\})). P' \{x/y\}) = T(F(x)).\]

\[P = \sum_{i=1}^{n} \alpha_i. P_i, \quad F = (\lambda y)(\sum_{i=1}^{n} \alpha_i. P_i). \quad \text{Then} \]
\[(v)(send_a(P \{x/y\}) | rec(v)) = (v)(send_a(\sum_{i=1}^{n} \alpha'_i. P_i \{x/y\}) | rec(v)) \]
(by lemma 3.4, \(\alpha'_i = \alpha_i \{x/y\}\)).

Also
\[T(F(x)) = T(\sum_{i=1}^{n} \alpha'_i. P_i \{x/y\}) = \sum_{i=1}^{n} \beta_i. Q_i, \]
where we distinguish five cases:

(i) \(\alpha_i\) is a first-order input, \(\alpha'_i = a(x_i)\). Then
\[
\beta_i = \alpha'_i = a(x_i) \quad \text{and} \quad Q_i = T(P_i \{x/y\}). \]

(ii) \(\alpha_i\) is a first-order output, \(\alpha'_i = \pi\beta\). Then
\[
\beta_i = \alpha'_i = \pi\beta \quad \text{and} \quad Q_i = T(P_i \{x/y\}). \]

(iii) \(\alpha_i\) is a higher-order output, \(\alpha'_i = \pi(X')\). Then
\[
\beta_i = \pi(x') \quad \text{and} \quad Q_i = T(P_i \{x/y\}). \]

(iv) \(\alpha_i\) is a higher-order output, \(\alpha'_i = \pi(G)\), \(G\) not a variable. Then
\[
\beta_i = \pi(u_i) \quad \text{for some fresh} \quad u_i, \quad Q_i = T(P_i \{x/y\}) | spawn_{w_i}(G). \]

(iv) \(\alpha_i\) is a higher-order input, \(\alpha'_i = a(V_i)\). Then
\[
\beta_i = a(x_i), \quad Q_i = T(P_i \{x/y\}). \]

By a series of deterministic \(\tau\)-transitions, we get
\[
(v)(send_a(\sum_{i=1}^{n} \alpha'_i. P_i \{x/y\}) | rec(v)) \]
\[
\Rightarrow (v_1 \ldots v_n w_1 \ldots w_{pq})(send_1 | \ldots | send_n | A_1 | \ldots | A_n | \Sigma_{pq}), \]
where, according to the four cases above, we get:

(i) \(\alpha_i\) is a first-order input, \(\alpha'_i = a(x_i)\). Then
The next transition is necessarily one of the following two:

(ii) \( \alpha_i \) is a first-order output, \( \alpha'_i = \overline{ab} \). Then
\[
\text{send}_i = \text{send}_i(P_i \{x/y\}), A_i = O(v_i abpq).
\]

(iii) \( \alpha_i \) is a higher-order output, \( \alpha'_i = \overline{\pi}(X') \). Then
\[
\text{send}_i = \text{send}_i(P_i \{x/y\}), A_i = O(v_i ax'pq).
\]

(iv) \( \alpha_i \) is a higher-order output, \( \alpha'_i = \overline{\pi}(G) \). Then
\[
\text{send}_i = \text{send}_i(P_i \{x/y\}) | \text{spawn}_{w_i} (G), A_i = O(v_i aw_i pq).
\]

(v) \( \alpha_i \) is a higher-order input, \( \alpha'_i = a(X_i) \). Then
\[
\text{send}_i = a(x_i). \text{send}_i(P_i \{x/y\}), A_i = I(v_i apq).
\]

This implies that
\[
(v)(\text{send}_a(\sum_{i=1}^n \alpha'_i P_i \{x/y\}) | \text{rec}(v)) \sim
\]
\[
(v_1...v_n w_1...w_n pq)(\text{send}_1 | ... | \text{send}_n | A_1 | ... | A_n | \text{Sem}(pq)).
\]

Now any transition executed by this agent will be a \( \tau \)-transition involving some \( A_i \), say \( A_k \), and \( \text{Sem}(pq) \), yielding
\[
(v_i \overline{a} v_i pq)(\text{send}_i | ... | \text{send}_n | A_1 | ... A_k' | ... A_n | \overline{\pi}(\text{Sem}(pq))
\]
where, in accordance with the five alternatives above, we have:

(i) \( \alpha_k \) is a first-order input, \( \alpha'_k = a(x_k) \). Then
\[
A_k' \sim a(x_k). \overline{\pi} x_k. \text{rec}(v_k) + q. I(v_i apq).
\]

(ii) \( \alpha_k \) is a first-order output, \( \alpha'_k = \overline{ab} \). Then
\[
A_k' \sim \overline{ab}. \text{rec}(v_k) + q. O(v_i abpq).
\]

(iii) \( \alpha_k \) is a higher-order output, \( \alpha'_k = \overline{\pi}(X') \). Then
\[
A_k' \sim \overline{\pi} x'. \text{rec}(v_k) + v. O(v_k ax'pq)
\]

(iv) \( \alpha_k \) is a higher-order output, \( \alpha'_k = \overline{\pi}(G) \), \( G \) not a variable. Then
\[
A_k' \sim \overline{\pi} \overline{\pi} x. \text{rec}(v_k) + v. O(v_k x'pq)
\]

(v) \( \alpha_k \) is a higher-order input, \( \alpha'_k = a(X_k) \). Then
\[
A_k' \sim a(x_k). \overline{\pi} x_k. \text{rec}(v_k) + q. I(v_i apq).
\]

The next transition is necessarily one of the following two:

1) A \( \tau \)-transition involving the channel \( q \) in \( A_k' \) and \( \overline{\pi}. \text{Sem}(pq) \), yielding again
\[
(v_i \overline{a} v_i pq)(\text{send}_i | ... | \text{send}_n | A_1 | ... | A_n | \overline{\pi}. \text{Sem}(pq));
\]

2) A transition \( \alpha \) executed by \( A_k' \) alone yielding
\[
(v_i \overline{a} v_i pq)(\text{send}_i | ... | \text{send}_n | A_1 | ... A_k' | ... A_n | \overline{\pi}. \text{Sem}(pq)) \sim
\]
\[
(v_i w_i pq)(\text{send}_i | ... | \text{send}_n | A_1 | ... A_k' | ... A_n | \overline{\pi}. \text{Sem}(pq)) \sim
\]

where, according to the five alternatives, we get:

(i) \( \alpha = a(x_k), A_k'' \sim \overline{\pi} x_k. \text{rec}(v_k), \)
\[
(v_i w_i)(\text{send}_k | A_k'') \sim (v_i)(a(x_i) \overline{\pi} \overline{\pi} x_i. \text{rec}(v_k)) \sim
\]
\[
\overline{\tau}(v_k)(\text{send}_i(P_k \{x/y\}) | \text{rec}(v_k)) \geq
\]
\[
(v_i)(\text{send}_i(P_k \{x/y\}) | \text{rec}(v_k)) \geq \overline{T}(P_k \{x/y\}) \text{ by induction.}
\]

Now, in this case \( \beta_k = a(x_k) \), and hence
\[
\overline{T}(F(x)) = \sum_{i=1}^n \beta_i. Q_i \overline{a(x_i)} Q_k = \overline{T}(P_k \{x/y\}),
\]

(ii) \( \alpha = \overline{ab}, A_k'' \sim \text{rec}(v_k), \)
\[
(v_i w_i)(\text{send}_k | A_k'') \sim (v_i)(\text{send}_i(P_k \{x/y\}) | \text{rec}(v_k)) \geq \overline{T}(P_k \{x/y\}) \text{ by induction.}
\]

In this case, \( \beta_k = \overline{ab} \), and hence
$$T(F(x)) = \sum_{i=1}^{n} \beta_i Q_i \xrightarrow{w} Q_k = T(P_k\{x/y\})$$

(iii) $$\alpha = \pi x, A_k'' \sim \text{rec}(v_k), (v_k w_k)(\text{send}_k | A''_k) \sim (v_k)(\text{send}_v(P_k\{x/y\}) | \text{rec}(v_k)) \geq T(P_k\{x/y\})$$

by induction.

In this case, $$\beta_k = \pi x,$$ and hence

$$T(F(x)) = \sum_{i=1}^{n} \beta_i Q_i \xrightarrow{w} Q_k = T(P_k\{x/y\}).$$

(iv) $$\alpha = \pi(w_k), A_k'' \sim \text{rec}(v_k); (v_k w_k)(\text{send}_k | A''_k) \sim (v_k)(\text{send}_v(P_k\{x/y\}) | \text{spawn}_{w_k}(G) \text{ rec}(v_k)) \geq T(P_k\{x/y\}) | \text{spawn}_{w_k}(G)$$

by induction.

In this case, $$\beta_k = \pi w_k,$$ and hence by induction

$$T(F(x)) = \sum_{i=1}^{n} \beta_i Q_i \xrightarrow{w} Q_k = T(P_k\{x/y\}) | \text{spawn}_{w_k}(G).$$

(v) $$\alpha = a(x_k), A_k'' \sim \pi x_k \text{ rec}(v_k)$$

By repeating an argument similar to case (i), we get $$(v_k w_k)(\text{send}_k | A''_k) \geq Q_k.$$ 

Thus, in all five cases $$(v_k w_k)(\text{send}_k | A''_k) \geq Q_k$$

On the other hand, if $$\sum_{i=1}^{n} \beta_i Q_i \xrightarrow{w} Q,$$

then necessarily $$\alpha = \pi x$$ for some $$k$$ such that $$1 \leq k \leq n,$$ and $$Q \equiv Q_k.$$

In this case we get

$$(v_1 \ldots v_n p_q)(\text{send}_1 | \ldots | \text{send}_n | A_1 | \ldots | A_n | \text{Sem}(p_q)) \xrightarrow{T}$$

$$(v_i \pi v_i(p_q)(\text{send}_1 | \ldots | \text{send}_n | A_1 | \ldots | A_n | \text{Sem}(p_q)) \xrightarrow{T}$$

$$(v_i \pi w_i(p_q)(\text{send}_1 | \ldots | \text{send}_n | A_1 | \ldots | A_n | \text{Sem}(p_q)) \xrightarrow{T}$$

$$(v_k w_k)(\text{send}_k | A''_k) \sim Q_k.$$
The last equivalence follows from the fact that neither of the two components of the agent may synchronize with any external agents before synchronizing with each other, upon which the continuation will not be affected by the restriction on \( v \).

Further

\[ !(v)(\overline{\text{m}}(v')) . \text{send}_a(P \{ x/y \}) | v'(v') . \text{rec}(v') \geq !T(P \{ x/y \}) = !T(P \{ x/y \}) \]

(by induction) = \( T(!P \{ x/y \}) = T(!P \{ x/y \}) = T(F(x)) \).

\( \star \)

\( P = (z)P', F = (\lambda y)(z)P', z \neq x. \) Then

\[ (v)(\text{send}_a(P \{ x/y \}) | \text{rec}(v)) = (v)(\text{send}_a((z)P' \{ x/y \}) | \text{rec}(v)) \sim \]

\[ (v)((z)\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \sim (z)(v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \geq \]

\[ (z)T(P' \{ x/y \}) \] (by induction) = \( T((z)P' \{ x/y \}) = T(F(x)) \).

\( \star \)

\( P = [a = b]P', F = (\lambda y)[a = b]P'. \) Then,

\[ (v)(\text{send}_a(P \{ x/y \}) | \text{rec}(v)) = (v)(\text{send}_a([a' = b']P' \{ x/y \}) | \text{rec}(v)) \sim \]

\[ (v)(\overline{\text{m}}_m \overline{\text{m}'}_m \overline{\text{m}''}_m \text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \sim \]

\[ \tau.\tau.\tau.(v)(\text{send}_a(P' \{ x/y \}) | [a' = b'] \text{rec}(v)) \geq \]

\[ (v)(\text{send}_a(P' \{ x/y \}) | [a' = b'] \text{rec}(v)) \sim \]

\[ [a' = b'](v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \]

since

\[ (v)(\text{send}_a(P' \{ x/y \}) | [a' = b'] \text{rec}(v)) \sim (v)(\text{send}_a(P' \{ x/y \}) \sim 0 \]

if \( a' \neq b', a' = a \{ x/y \}, b' = b \{ x/y \}, \) otherwise

\[ (v)(\text{send}_a(P' \{ x/y \}) | [a' = b'] \text{rec}(v)) \sim (v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \sim \]

\[ [a' = b'](v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \]

Further,

\[ [a' = b'](v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \geq [a' = b'] \overline{\text{T}}(P' \{ x/y \}) \] (by induction)

\[ = \overline{T}[a' = b']P' \{ x/y \} = \overline{T}(F(x)) \]

\( \star \)

\( P = P' \setminus z, F = (\lambda y)(P' \setminus z). \) Then,

\[ (v)(\text{send}_a(P \{ x/y \}) | \text{rec}(v)) = (v)(\text{send}_a(P' \{ x/y \}) \setminus z') | \text{rec}(v) \] (z' = z \{ x/y \})

\[ \sim (v)(\overline{\text{m}}_b \overline{\text{m}'_b} \text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \sim \]

\[ \tau.\tau.(v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \sim \]

\[ (v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \sim \]

\[ ((v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \setminus z') \]

since \( z \) is never the subject of any action executed by the agent \( \text{send}_a(P' \{ x/y \}) \).

Further,

\[ ((v)(\text{send}_a(P' \{ x/y \}) | \text{rec}(v)) \setminus z') \geq (\overline{T}(P' \{ x/y \})) \setminus z' \] (by induction)

\[ = \overline{T}(P' \{ x/y \} \setminus z') = \overline{T}(F(x)) \]

\( \star \)

\( P = Y'(a'), F = (\lambda y)(Y' \{ a' \}). \) Then

\[ (v)(\text{send}_a(P \{ x/y \}) | \text{rec}(v)) = (v)(\text{send}_a(Y' \{ a' \} \{ x/y \}) | \text{rec}(v)) = \]

\[ T(Y' \{ x/y \}) = T(F(x)) \]

\( \star \)

\( P = Y'(X'), F = (\lambda y)(Y' \{ X' \}). \) Then

\[ (v)(\text{send}_a(P \{ x/y \}) | \text{rec}(v)) = (v)(\text{send}_a(Y' \{ X' \}) | \text{rec}(v)) \sim \]

\[ (v)(y'(u) \text{m} \text{m}' \text{m}'' x. 0 | \text{rec}(v)) \sim \]

\[ y'(u), \text{m}(v), \text{m} x. \text{rec}(v) = \overline{T}(Y' \{ X' \}) = \overline{T}(F(x)) \]

\( \star \)

\( P = Y'(G), F = (\lambda y)(Y' \{ G \}), G \) not a variable. Then

\[ (v)(\text{send}_a(P \{ x/y \}) | \text{rec}(v)) = (v)(\text{send}_a(Y' \{ G \} \{ x/y \}) | \text{rec}(v)) \sim \]

\[ (v)(y'(u) \text{m} \text{m}' \text{m}'' x. (\text{spawn}_{a'}(G \{ x/y \}) | \text{rec}(v)) \sim \]

\[ y'(u), \text{m}(v), \text{m} x. (\text{spawn}_{a'}(G \{ x/y \}) | \text{rec}(v)) = \]

\[ \overline{T}(Y' \{ X' \}) = \overline{T}(F(x)) \]

\[ = \overline{T}(F(x)) \]
\[ T(Y' \langle G \{ x/y \} \rangle) = T(F \langle x \rangle). \]

**Proof of (ii):** \((w)(\text{spawn}_w(F) \mid T(W(G))) \cong_{ \tau } T(F(G))\), where \(F = (\lambda Y)P\).

If \(G = X\) for some process variable \(X\), the proof is similar to the case above, noting that in this case, by lemma 3.4

\[ T(P) \{ x/y \} = T(P \{ X/Y \}), \quad \text{send}_w(P) \{ x/y \} = \text{send}_w(P) \{ X/Y \} \]

and \(\text{spawn}_w(G) \{ x/y \} = \text{spawn}_w(G) \{ X/Y \}\).

Therefore we assume that \(G\) is not a process variable.

The proof is by induction on \(P\)’s formation and \(F\)’s abstraction order. For \(F\) of order 1, that is, \(F = (\lambda Y)P\), the proof is part (i) above.

We prove that \((vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \geq T(F(G))\), since then

\[ \tau.\tau.(wuv)(\text{spawn}_w((\lambda Y)P) \mid T(W(G))) = \]

\[ (w)(\text{spawn}_w((\lambda Y)P) \mid w(u).u(v).u(y).\text{send}_w(P) \mid \overline{m}(u).\overline{m}(v).\overline{m}(x).\text{rec}(v) \mid \text{spawn}_x(G)) \sim \]

\[ \tau.\tau.(wuv)(\text{spawn}_w((\lambda Y)P) \mid \text{send}_w(P) \{ x/y \} \mid \text{rec}(v) \mid \text{spawn}_x(G)) \geq \]

\[ (vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

(by lemma 3.4 and since \(w \notin \text{fin}(\text{send}_v(P \{ x/y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \)

\[ \geq T(F(G)) \] as desired.

\[ \bullet P = 0. \quad \text{Then} \]

\[ (vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) = \]

\[ (vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) = \]

\[ (vx)(\tau.\tau.(v)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \sim (v)(\tau.\tau.0 \mid \text{rec}(v)) \sim \tau.\text{rec}(0) \geq 0 = T(0) = T(0 \langle x \rangle 0 \langle y \rangle) = T(F(G)). \]

\[ \bullet P = \overline{a}b.P', \quad F = (\lambda Y)\overline{a}b.P'. \]

Then

\[ (vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) = \]

\[ (vx)(\text{send}_v(\overline{a}b.P' \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) = \]

\[ \tau.\tau.(v)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq \overline{a}b.(vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq (a)(vx)(\text{send}_v(P' \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq (a).\overline{a}b.T(P' \{ G/Y \}) \] (by induction)

\[ = T(\overline{a}b.P' \{ G/Y \}) = T(F(G)). \]

\[ \bullet P = a(b).P', \quad F = (\lambda Y)a(b).P'. \]

Then

\[ (vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) = \]

\[ (vx)(\tau.\tau.(v)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq a(b).(vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq (a)(vx)(\text{send}_v(P' \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq (a).\overline{a}b.T(P' \{ G/Y \}) \] (by induction)

\[ = T(a(b).P' \{ G/Y \}) = T(F(G)). \]

\[ \bullet P = a(X').P', \quad F = (\lambda Y)a(X').P', \quad \text{where} \ X' \neq X, \ X' \neq Y. \quad \text{Similar.} \]

\[ \bullet P = \overline{a}(X').P', \quad F = (\lambda Y)\overline{a}(X').P'. \]

Then

\[ (vx)(\text{send}_v(P \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) = \]

\[ (vx)(\text{send}_v(\overline{a}(X').P' \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \sim \]

\[ \geq \overline{a}(X').\overline{a}(X').(vx)(\text{send}_v(P' \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq (a)(vx)(\text{send}_v(P' \{ X/Y \}) \mid \text{rec}(v) \mid \text{spawn}_x(G)) \]

\[ \geq (a).\overline{a}(X').\overline{a}(X').T(P' \{ G/Y \}) \] (by induction)

\[ = T(a(X').P' \{ G/Y \}) = T(F(G)). \]

\[ \bullet P = a(X').P', \quad F = (\lambda Y)a(X').P', \quad \text{where} \ X' \neq X, \ X' \neq Y. \quad \text{Similar.} \]
\( (\forall x)(\forall \alpha, \beta \alpha^x \{x/y\}, (send\alpha(P^x \{X/Y\}) | rec\beta | spawn\alpha(G)) \sim \\
\tau.\tau.\tau.(\forall x)(send\alpha(P^x \{X/Y\}) | \alpha^x \{x/y\}, rec\beta | spawn\alpha(G)) \geq \\
(\forall x)(send\alpha(P^x \{X/Y\}) | \alpha^x \{x/y\}, rec\beta | spawn\alpha(G)) \sim \\
\alpha^x \{x/y\}. (\forall x)(send\alpha(P^x \{X/Y\}) | rec\beta | spawn\alpha(G)) \geq \\
\alpha^x \{x/y\}. T(P^x \{X/Y\}) \text{ (by induction)} = T(\alpha(\{X'/X\})). P' \{X/Y\} = T(F(G)).

\( P = \alpha(G'), P^x, F = (\lambda Y)\alpha(G'), P^x, G' \text{ not a process variable.} \)

Then
\( (\forall x)(send\alpha(P^x \{X/Y\}) | rec\beta | spawn\alpha(G)) = \\
(\forall x)(send\alpha(\alpha(G') \{X/Y\}, P^x \{X/Y\}) | rec\beta | spawn\alpha(G)) \sim \\
(\forall x)(\forall \alpha, \beta \alpha^x \{x/y\}, (send\alpha(P^x \{X/Y\}) | \alpha^x \{x/y\}, rec\beta | spawn\alpha(G)) \sim \\
\tau.\tau.(\forall x)(\forall \alpha, \beta \alpha^x \{x/y\}, rec\beta | spawn\alpha(G)) \geq \\
(\forall x)(\forall \alpha, \beta \alpha^x \{x/y\}, rec\beta | spawn\alpha(G)) \geq \\
\alpha^x \{x/y\}. (\forall x)(\forall \alpha, \beta \alpha^x \{x/y\}, rec\beta | spawn\alpha(G)) \geq \\
\alpha^x \{x/y\}. T(P^x \{X/Y\}) | \alpha^x \{x/y\}, rec\beta | spawn\alpha(G)) \geq \\
\alpha^x \{x/y\}. T(\alpha(\{G'/G\})). P^x \{G'/G\} = T(F(G)).

\[ P = \sum_{i=1}^n \alpha_i, P_i, F = (\lambda Y)(\sum_{i=1}^n \alpha_i, P_i). \text{ Then} \\
(\forall x)(send\alpha(P^x \{X/Y\}) | rec\beta | spawn\alpha(G)) = \\
(\forall x)(send\alpha(\sum_{i=1}^n \alpha_i \{X/Y\}, P_i \{X/Y\}) | rec\beta | spawn\alpha(G)).

Also
\[ T(F(G)) = T(\sum_{i=1}^n (\alpha_i, P_i) \{G/Y\} = \sum_{i=1}^n \beta_i. Q_i, \]

where we distinguish five cases:

(i) \( \alpha_i \) is a first-order input. Then \( \beta_i = \alpha_i, \text{ and } Q_i = T(P_i \{G/Y\}). \)

(ii) \( \alpha_i \) is a first-order output. Then \( \beta_i = \alpha_i \text{ and } Q_i = T(P_i \{G/Y\}). \)

(iii) \( \alpha_i \) is a higher-order output, \( \alpha_i = \alpha(X'), X' \neq Y. \text{ Then} \)
\[ \beta_i = \alpha(\alpha^x) \text{ and } Q_i = T(P_i \{G/Y\}). \]

(iv) \( \alpha_i \) is a higher-order output, \( \alpha_i = \alpha(G'), G' \text{ not a variable. Then} \)
\[ \beta_i = \alpha(w_i) \text{ for some fresh } w_i, Q_i = T(P_i \{G/Y\}) | spawn_{w_i}(G' \{G/Y\}). \]

(v) \( \alpha_i \) is a higher-order input, \( \alpha_i = a(X_i). \text{ Then} \)
\[ \beta_i = a(x_i), Q_i = T(R_i \{G/Y\}). \]

By a series of deterministic \( \tau \)-transitions, we get
\( (\forall x)(send\alpha(\sum_{i=1}^n \alpha_i \{X/Y\}, P_i \{X/Y\}) | rec\beta | spawn\alpha(G)) \Rightarrow \sim \\
(v_1 \ldots v_n \pi x p q)(send \ldots | send \ldots | A_1 \ldots \ A_n | sem(p q) | spawn\alpha(G))

where, in accordance with the five cases above, we get:

(i) \( \alpha_i \) is a first-order input, \( \alpha_i = a(x_i). \text{ Then} \)
\[ send_i = v_i(x_i).send\alpha_i(P_i \{X/Y\}), A_i = I(v_i ap q). \]

(ii) \( \alpha_i \) is a first-order output, \( \alpha_i = \alpha x \).
\[ send_i = send\alpha_i(P_i \{X/Y\}), A_i = O(v_i ap q). \]

(iii) \( \alpha_i \) is a higher-order output, \( \alpha_i = \alpha(X'). \text{ Then} \)
\[ send_i = send\alpha_i(P_i \{X/Y\}), A_i = O(v_i ax p q). \]

(iv) \( \alpha_i \) is a higher-order output, \( \alpha_i = \alpha(G'). \text{ Then} \)
\[ send_i = send\alpha_i(P_i \{X/Y\}) | spawn_{w_i}(G' \{X/Y\}), A_i = O(v_i aw p q). \]
(v) $\alpha_i$ is a higher-order input, $\alpha_i' = a(X_i)$. Then

$$send_i = a(x_i).send_{v_i}(P_i \{ X/Y \}), A_i = I(v_i apq).$$

This implies that

$$(\forall \alpha_i(\sum_{i=1}^n \alpha_i \{ X/Y \}, P_i \{ X/Y \} \mid rec \langle v \rangle \mid spawn \langle G \rangle) \sim$$

$$(v_1 \ldots v_n w_1 \ldots w_n a \langle pq \rangle(snd_i \mid \ldots \mid snd_n \mid A_1 \mid \ldots \mid A_n \mid Sem \{ pq \} \mid spawn \langle G \rangle)).$$

Now any transition executed by this agent will be a $\tau$-transition involving some $A_i$, say $A_k$, and $Sem \{ pq \}$, yielding

$$(v_i w_i pq)(snd_1 \mid \ldots \mid snd_n \mid A_1 \mid \ldots \mid A_k \mid \ldots \mid A_n \mid \overline{\tau Sem \{ pq \}})$$

where, in accordance with the five alternatives above, we have:

(i) $\alpha_k$ is a first-order input, $\alpha_k = a(x_k)$. Then

$$A_k' \sim a(x_k).\overline{\tau}v_k.rec(v_k) + q.I(v_k apq).$$

(ii) $\alpha_k$ is a first-order output, $\alpha_k = \overline{v}_k$. Then

$$A_k' \sim \overline{v}_k.rec(v_k) + q.O(v_k apq).$$

(iii) $\alpha_k$ is a higher-order output, $\alpha_k = \overline{\pi}(X')$. Then

$$A_k' \sim \overline{\pi}w_k.rec(v_k) + q.O(v_k apq).$$

(iv) $\alpha_k$ is a higher-order output, $\alpha_k = \overline{\pi}(G')$. Then

$$A_k' \sim \overline{\pi}w_k.rec(v_k) + v.O(v_k apq).$$

(v) $\alpha_k$ is a higher-order input, $\alpha_k = a(X_k)$. Then

$$A_k' \sim a(x_k).\overline{\tau}v_k.xk.rec(v_k) + q.I(v_k apq).$$

The next transition is necessarily one of the following two:

1) a $\tau$-transition involving the channel $q$ in $A_k'$ and $\tau\overline{\tau}Sem\{pq\}$, yielding again

$$(v_i w_i pq)(snd_1 \mid \ldots \mid snd_n \mid A_1 \mid \ldots \mid A_n \mid \tau Sem\{pq\} \mid spawn\langle G \rangle);$$

2) a transition $\alpha$ executed by $A_k'$ alone yielding

$$(v_i w_i pq)(snd_1 \mid \ldots \mid snd_n \mid A_1 \mid \ldots \mid A_k' \mid \ldots \mid A_n \mid \tau Sem\{pq\} \mid spawn\langle G \rangle) \sim$$

$$(v_k w_k x)(snd_1 \mid \ldots \mid snd_n \mid A_k'' \mid spawn\langle G \rangle)$$

where, according the the five alternatives, we get:

(i) $\alpha = a(x_k), A_k'' \sim \overline{\tau}v_k.xk.rec(v_k),$

$$(v_i w_i x)(send_k \mid A_k'' \mid spawn\langle G \rangle) \sim$$

$$(v_k(t_{v_k}(x_k).send(v_k(P_k \{ X/Y \})) \mid \overline{\tau}v_k.xk.rec(v_k) \mid spawn\langle G \rangle) \sim$$

$$\tau(v_k x)(send(v_k(P_k \{ X/Y \})) \mid rec(v_k) \mid spawn\langle G \rangle) \geq$$

$$(v_k x)(send(v_k(P_k \{ X/Y \})) \mid rec(v_k) \mid spawn\langle G \rangle) \geq$$

$$T(P_k \{ G/Y \})$$

by induction.

Now, in this case $\beta_k = a(x_k)$, and hence

$$T(F(G)) = \sum_{i=1}^n \beta_i Q_i \xrightarrow{a(x_k)} Q_k = T(P_k \{ G/Y \}).$$

(ii) $\alpha = \overline{v}_k, A_k'' \sim rec\langle v \rangle;$

$$(v_k w_k x)(send_k \mid A_k'' \mid spawn\langle G \rangle) \sim$$

$$(v_k(t_{send}(v_k(P_k \{ X/Y \})) \mid rec\langle v \rangle \mid spawn\langle G \rangle)$$

In this case, $\beta_k = \overline{v}_k$, and hence

$$T(F(G)) = \sum_{i=1}^n \beta_i Q_i \xrightarrow{\overline{v}_k} Q_k = T(P_k \{ G/Y \}).$$

(iii) $\alpha = \overline{\pi}X', A_k'' \sim rec\langle v \rangle$: similar.

(iv) $\alpha = \overline{\pi}w_k, A_k'' \sim rec\langle v \rangle;$

$$(v_k w_k x)(send_k \mid A_k'' \mid spawn\langle G \rangle) \sim$$

$$(v_k(t_{send}(v_k(P_k \{ X/Y \})) \mid spawn\langle G' \{ X/Y \} \rangle \mid rec\langle v \rangle \mid spawn\langle G \rangle) \geq$$

$$T(P_k \{ G/Y \}) \mid spawn\langle G \{ X/Y \} \rangle$$

by induction.
In this case, $\beta_k = \overline{n}(w_k)$, and hence
\[
\mathcal{T}(F(G)) = \sum_{i=1}^n \beta_i Q_i \overline{n}(w_k) \rightarrow Q_k = T(P_k \{X/Y\}) | \text{spawn}_{w_k}(G \{X/Y\}).
\]

(v) $a = a(x_k), A_k' \sim \overline{n}x_k \text{rec}(v_k)$

By repeating an argument similar to case (i), we get
\[(v_k w_k x)(\text{send}_k | A_k' | \text{spawn}_{x}(G)) \geq Q_k.
\]

Thus, in all five cases,
\[(v_k w_k x)(\text{send}_k | A_k' | \text{spawn}_{x}(G)) \geq T(P_k \{X/Y\}) = Q_k
\]

On the other hand, if
\[\sum_{i=1}^n \beta_i Q_i \overline{n} \rightarrow Q,
\]

then necessarily $a = \beta_k$ for some $k$ such that $1 \leq k \leq n$, and $Q \equiv Q_k$.

In this case, we get also
\[(w_1 \ldots w_n x_p q)(\text{send}_1 | \ldots | \text{send}_n | A_1 | \ldots | A_n | \text{Sem}(pq) | \text{spawn}_{x}(G)) \overline{n} \rightarrow
\]
\[(v_1 \overline{n} x_p q)(\text{send}_1 | \ldots | \text{send}_n | A_1 | \ldots | A_n | \text{Sem}(pq) | \text{spawn}_{x}(G)) \overline{\beta} \rightarrow
\]
\[(v_1 v_2 \overline{n} x_p q)(\text{send}_1 | \ldots | \text{send}_n | A_1 | \ldots | A_n | \text{Sem}(pq) | \text{spawn}_{x}(G)) \overline{\sim} Q_k.
\]

Therefore, by pairing together all agents that may be reached from
\[(v_1 \ldots v_n pq)(\text{send}_1 | \ldots | \text{send}_n | A_1(\overline{u}_1) | \ldots | A_n(\overline{u}_n) | \text{Sem}(pq))
\]

by means of $\tau$-transitions with $T(F(G)) = \sum_{i=1}^n \beta_i Q_i$, and $(v_k(\text{send}_k) | A_k')$ with $Q_k$, as well as all pairs of agents in expansions of $(v_k(\text{send}_k) | A_k')$ and $Q_k$, we get an expansion $\varepsilon$ such that $(T(F(G)), (w)(\text{spawn}_w(F) | T(W(G)))) \in \varepsilon$.

Hence $T(F(G)) \sim (w)(\text{spawn}_w(F) | T(W(G)))$.

$P = P_1 \mid P_2, F = (\lambda Y)((P_1 \mid P_2)$. Then
\[(\overline{x}r)(\text{send}_1(P_1 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) =
\]
\[(\overline{x}r)(\text{send}_2(P_1 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) =
\]
\[\tau, \tau, \tau (v_1 v_2 x)((\text{send}_1(P_1 \{X/Y\}) | \text{send}_2(P_2 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G))
\]

by lemma 3.3(\overline{u}), and since
\[v_1 \notin n(\text{send}_2(P_2 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) and
\]
\[v_2 \notin n(\text{send}_1(P_1 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)).
\]

Further
\[(v_1 v_2 x)(\text{send}_1(P_1 \{x/y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) | (v_2 x)(\text{send}_2(P_2 \{x/y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) \geq T(P_1 \{G/Y\}) | P_2 \{G/Y\} \text{ (by induction) } = T((P_1 \mid P_2) \{G/Y\}) = T(F(G)).
\]

$P = !P', F = (\lambda Y)!P'. Then$
\[(\overline{x}r)(\text{send}_1(P_1 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) =
\]
\[(\overline{x}r)(\text{send}_2(P_1 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) =
\]
\[\tau, \tau, \tau (v_1 v_2 x)(\text{send}_1(P_1 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) \geq
\]
\[(v_1 v_2 x)(\text{send}_2(P_2 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) \geq
\]
\[!\overline{x}r (P_1 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)) \geq
\]
\[!\overline{x}r (P_2 \{X/Y\}) | \text{rec}(\overline{v}) | \text{spawn}_{x}(G)).
\]

247
Further
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \geq \]
\[ \forall z. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \sim \]
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \sim \]
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \sim \]

Further
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \geq \]
\[ (z) T(P^x \{ G/Y \}) \] (by induction) = \[ T((z) P^x \{ G/Y \}) = T(F(G)). \]

Further
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \geq \]
\[ (z) T(P^x \{ G/Y \}) \] (by induction) = \[ T((z) P^x \{ G/Y \}) = T(F(G)). \]

Further
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \geq \]
\[ (z) T(P^x \{ G/Y \}) \] (by induction) = \[ T((z) P^x \{ G/Y \}) = T(F(G)). \]

Further
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \geq \]
\[ (z) T(P^x \{ G/Y \}) \] (by induction) = \[ T((z) P^x \{ G/Y \}) = T(F(G)). \]

Further
\[ \forall v. r. (send_u(P^x \{ X/Y \}) | rec(w) | spawn_w(G)) \geq \]
\[ (z) T(P^x \{ G/Y \}) \] (by induction) = \[ T((z) P^x \{ G/Y \}) = T(F(G)). \]
\[\forall(u).\forall(v).\forall(x'.(\text{rec}(v) | \{x\rbrack_{\text{spawn}}(G)) \sim \]
\[\forall(u).\forall(v).\forall(x'.\text{rec}(v) = \\
T(Y(x')) = T(Y(x')\{G/X\}) = T(F(G)).\]

Assume now that \(X = Y\). Then
\[P = X(x'), F = (\lambda X).X(x').\]
Hence,
\[(vx)(send_\pi(P\{X/Y\}) | \text{rec}(v) | \text{spawn}_\pi(G)) = \]
\[(vx)(send_\pi(X(x')) | \text{rec}(v) | \text{spawn}_\pi(G)) \sim \]
\[(vx)(\pi(u).\pi(v).\pi(x'.(\text{rec}(v) | x(u).u(v).u(x').send_\pi(G(x')) | \text{spawn}_\pi(G)) \sim \]
\[\tau.\tau.\tau.(vx)(\text{rec}(v) | send_\pi(G(x')) | \text{spawn}_\pi(G)) \geq \]
\[(vx)(\text{rec}(v) | send_\pi(G(x')) | \text{spawn}_\pi(G)) \sim \]
\[(v)(\text{rec}(v) | send_\pi(G(x')))) \geq \]
\[T(G(x')) \quad \text{(by (i))} = T(F(G)).\]

\(\bullet P = Y(X'), F = (\lambda X)Y((X'))\).

We may distinguish three cases:

(i) \(Y \neq X, X' \neq X\). Then
\[(vx)(send_\pi(P\{X/Y\}) | \text{rec}(v) | \text{spawn}_\pi(G)) = \]
\[(vx)(\pi(u).\pi(v).\pi(x'.(\text{rec}(v) | \text{spawn}_\pi(G)) \sim \]
\[\forall(u).\forall(v).\forall(x'.\text{rec}(v) | (x)\text{spawn}_\pi(G)) \sim \]
\[\forall(u).\forall(v).\forall(x'.\text{rec}(v) = \\
T(Y(x')) = T(Y(x')\{G/X\}) = T(F(G)).\]

(ii) \(Y \neq X, X' = X\). Then \(F = (\lambda X)Y(X)\). Then
\[(vx)(send_\pi(P\{X/Y\}) | \text{rec}(v) | \text{spawn}_\pi(G)) = \]
\[(vx)(\pi(u).\pi(v).\pi(x'.(\text{rec}(v) | \text{spawn}_\pi(G)) \sim \]
\[\forall(u).\forall(v).\forall(x'.(\text{rec}(v) | \text{spawn}_\pi(G)) = \\
T(Y(G)) = T(Y(X)\{G/X\}) = T(F(G)).\]

(iii) \(Y = X\). Then \(F = (\lambda X)X(X')\), \(X' \neq X\) if agent is well-sorted.

Since \(G\) is not a variable, \(G \equiv (\lambda y')Q\) or \((\lambda Y'\)Q for some agent \(Q\) and name \(y'\) or agent variable \(Y'\).

Then
\[(vx)(send_\pi(P\{X/Y\}) | \text{rec}(v) | \text{spawn}_\pi(G)) = \]
\[(vx)(send_\pi(X(x')) | \text{rec}(v) | \text{spawn}_\pi(G)) \sim \]
\[(vx)(\pi(u).\pi(v).\pi(x'.(\text{rec}(v) | x(u).u(v).u(y').send_\pi(Q) | \text{spawn}_\pi(G)) \sim \]
\[\tau.\tau.\tau.(vx)(\text{rec}(v) | send_\pi(Q) | \text{spawn}_\pi(G)) \geq \]
\[(vx)(\text{rec}(v) | send_\pi(Q) | \text{spawn}_\pi(G)) \geq \\
(v)(\text{rec}(v) | send_\pi(Q) \{x'/y'\}) \]

since we may assume \(x \notin \text{fn}(\text{rec}(v) | send_\pi(Q) \{x'/y'\}).\)

Now
\[(v)(\text{rec}(v) | send_\pi(Q) \{x'/y'\}) \geq T(G(X')) \]

by induction on order of abstraction since \(G\) has lower order than \(F\) if \(F(G)\) is well-sorted, and by (i) and remark at the beginning of the proof of (ii) since \(X'\) is a process variable.

\(\bullet P = Y(F'), F = (\lambda X)Y(F')\), \(F'\) not a variable.

Assume \(Y \neq X\). Then
\[(vx)(send_\pi(P\{X/Y\}) | \text{rec}(v) | \text{spawn}_\pi(G)) = \]
\[(vx)(send_\pi(Y(F')) | \text{rec}(v) | \text{spawn}_\pi(G)) \sim \]

249
By induction and lemma 3.8, for every channel $v$

\[ \mathcal{T}(Y(F') \{ G/X \}) = \mathcal{T}(F(G)). \]

Assume now $X = Y$. Then

\[ P = X(F'), F = (\lambda X)X((F')). \]

Now, $X$ does not occur free in $F'$ if $P$ is well-sorted.

Hence,

\[ (v)(\pi(u) . \pi(w') . \text{spawn}_w(F') \mid \text{rec}(v) \mid \text{spawn}_w(G)) = \]

\[ (v)(\pi(u) . \pi(w') . \text{spawn}_w(F') \mid \text{rec}(v) \mid \text{spawn}_w(G)) \]

\[ \sim \mathcal{T}(G(F') = \mathcal{T}(F(G)) \]

by induction, since $G$ has lower order than $F$ if $F(G)$ is well-sorted.

**Lemma 3.10:** For any process $P \in \Pi B$ and abstraction $F$ that is not a process variable,

\[ \mathcal{T}(P(F/W)) \leq (w)(\text{spawn}_w(F) \mid \mathcal{T}(P)). \]

**Proof:** By structural induction on $P$'s formation.

- $P \equiv 0$. Then

\[ (w)(\mathcal{T}(P) \mid \text{spawn}_w(F)) = (w)(\mathcal{T}(0) \mid \text{spawn}_w(F)) = \]

\[ (w)(0 \mid \text{spawn}_w(F)) \sim (w)(\text{spawn}_w(F)) \sim 0 = \mathcal{T}(0) = \mathcal{T}(0(F/W)) = \]

\[ (w)(\mathcal{T}(P) \mid \text{spawn}_w(F)). \]

- $P = P_1 \parallel P_2$. Then

\[ \mathcal{T}(P(F/W)) = \mathcal{T}(P_1 \{ F/W \} \parallel P_2 \{ F/W \}) = \]

\[ \mathcal{T}(P_1 \{ F/W \}) \parallel \mathcal{T}(P_2 \{ F/W \}) \leq \]

\[ (w)(\text{spawn}_w(F) \mid \mathcal{T}(P_1)) \parallel (w)(\text{spawn}_w(F) \mid \mathcal{T}(P_2)) \text{ (by induction) } \sim \]

\[ (w)(\text{spawn}_w(F) \mid \mathcal{T}(P_1)) \parallel (w)(\text{spawn}_w(F) \mid \mathcal{T}(P_2)) \text{ (by lemma 3.3(ii)) } = \]

\[ (w)(\text{spawn}_w(F) \mid \mathcal{T}(P_1 \parallel P_2)) \sim (w)(\text{spawn}_w(F) \mid \mathcal{T}(P)). \]

- $P = \pi_y. P'$. Then

\[ \mathcal{T}(P(F/W)) = \mathcal{T}(\pi_y. P'(F/W)) = \pi_y. \mathcal{T}(P'(F/W)) \leq \]

\[ \pi_y. (w)(\text{spawn}_w(F) \mid \mathcal{T}(P')) \text{ (by induction) } \sim \]

\[ (w)(\text{spawn}_w(F) \mid \pi_y. \mathcal{T}(P')) \text{ (by expansion) } = \]

\[ (w)(\text{spawn}_w(F) \mid \mathcal{T}(\pi_y. P')) \sim (w)(\text{spawn}_w(F) \mid \mathcal{T}(P)). \]

- $P = x(y). P'$.

By induction and lemma 3.4, for every channel $z$

\[ \mathcal{T}(P(F/W)) \{ z/y \} = \mathcal{T}(P'(F/W)) \{ z/y \} \leq \]

\[ (w)(\text{spawn}_w(F(z/y)) \mid \mathcal{T}(P'(z/y))) = (w)(\text{spawn}_w(F) \mid \mathcal{T}(P')) \{ z/y \}. \]

Hence
\[ T(P \{ F/W \}) = T(x(y), P \{ F/W \}) =
\]
\[ x(y). T(P \{ F/W \}) \leq x(y). (w)(\text{spawn}_w(F) \mid T(P')) \] (by induction)
\[ = (w)(\text{spawn}_w(F) \mid x(y). T(P')) \] (by expansion)
\[ = (w)(\text{spawn}_w(F) \mid T(x(y).P')) = (w)(\text{spawn}_w(F) \mid T(P)). \]

\[ \bullet P = x(Y).P', Y \neq W. \]

By induction and lemma 3.4, for any process variable \( Z \)
\[ T(P' \{ F/W \}) \{ z/y \} = T(P' \{ Z/Y \} \{ F\{Z/Y\}/W \}) \leq
\]
\[ (w)(\text{spawn}_w(F) \{ Z/Y \} \mid T(P' \{ Z/Y \})) =
\]
\[ (w)(\text{spawn}_w(F) \mid T(P')) \{ z/y \}. \]

Hence
\[ T(P \{ F/W \}) = T(x(Y), P' \{ F/W \}) = x(y). T(P' \{ F/W \}) \leq
\]
\[ x(y). (w)(\text{spawn}_w(F) \mid T(P')) \] (by induction)
\[ \sim (w)(\text{spawn}_w(F) \mid x(y). T(P')) \] (by expansion)
\[ = (w)(\text{spawn}_w(F) \mid T(x(Y).P')) = (w)(\text{spawn}_w(F) \mid T(P)). \]

\[ \bullet P = \pi(X'). P'. \]

Assume first that \( X' \neq W \). Then
\[ T(P \{ F/W \}) = T((\pi(X'). P') \{ F/W \}) =
\]
\[ \pi(X'). T(P \{ F/W \}) \leq
\]
\[ \pi(X'). (w)(\text{spawn}_w(F) \mid T(P')) \] (by induction)
\[ \sim (w)(\text{spawn}_w(F) \mid \pi(X'). T(P')) \] (by expansion)
\[ = (w)(\text{spawn}_w(F) \mid T(\pi(X').P')) \sim (w)(\text{spawn}_w(F) \mid T(P)). \]

Assume now \( X' = W \). Then
\[ T(P \{ F/W \}) = T((\pi(W). P') \{ F/W \}) =
\]
\[ \pi(w). T(P' \{ F/W \}) \leq
\]
\[ \pi(w). ((w)(\text{spawn}_w(F) \mid T(P')) \mid \text{spawn}_w(F)) \] (by induction)
\[ \sim \pi(w). (w)(\text{spawn}_w(F) \mid T(P')) \mid \text{spawn}_w(F) \] (by lemma 3.3(i))
\[ \sim \pi(w). (w)(\text{spawn}_w(F) \mid T(P')) \mid \text{spawn}_w(F) \] (by expansion)
\[ = (w)(\text{spawn}_w(F) \mid T(\pi(W).P')) = (w)(\text{spawn}_w(F) \mid T(P)). \]

\[ \bullet P = \pi(G). P', G \text{ not a variable}. \]

Then
\[ T(P \{ F/W \}) = T((\pi(G). P') \{ F/W \}) =
\]
\[ \pi(G). T(P \{ F/W \}) \leq
\]
\[ \pi(G). ((w)(\text{spawn}_w(F) \mid T(P')) \mid \text{spawn}_w(G\{F/W\})) \] (by induction)
\[ \sim \pi(G). (w)(\text{spawn}_w(F) \mid T(P')) \mid \text{spawn}_w(G\{F/W\})) \] (since \( w \notin \text{fin(}\text{spawn}_w(G\{F/W\})\))
\[ \sim \pi(G). (w)(\text{spawn}_w(F) \mid T(P')) \mid \text{spawn}_w(G) \] (by lemma 3.8 Cor. 2)
\[ \sim (w)(\text{spawn}_w(F) \mid \pi(G). T(P')) \mid \text{spawn}_w(G)) \] (by expansion)
\[ = (w)(\text{spawn}_w(F) \mid T(\pi(G).P')) = (w)(\text{spawn}_w(F) \mid T(P)). \]

\[ \bullet P = \sum_{i=1}^{n} \alpha_i. P_i. \]

By induction, for \( i = 1, 2, \ldots, n \),
\[ T((\alpha_i. P_i) \{ F/W \}) \leq (w)(\text{spawn}_w(F) \mid T(\alpha_i. P_i)) = (w)(\text{spawn}_w(F) \mid \beta_i.Q_i) \]
where
\[ \alpha_i = \exists x' \Rightarrow \beta_i = \exists x', Q_i = T(P_i) \]
\[ \alpha_i = \exists(G) \Rightarrow \beta_i = \exists(w'), Q_i = (T(P_i) \mid \text{spawn}_w(G)) \]
\[ (G \text{ not a variable}) \]
\[ \alpha_i = \exists y \Rightarrow \beta_i = \exists y, Q_i = T(P_i) \]
\[ \alpha = x(y) \text{ or } x(Y) \Rightarrow \beta_i = x(y), Q_i = T(P_i) \]

Now \((w)(\text{spawn}_w(F) \mid \beta_i, Q_i)\) is strongly equivalent to a well-guarded agent, either \(\beta_i, (w)(\text{spawn}_w(F) \mid Q_i)\) if \(\beta_i \neq \exists w\) for any channel \(x\), otherwise \(\exists(w)(\text{spawn}_w(F) \mid Q_i)\). Since \(\leq\) is preserved by well-guarded sums, and since all agents in the sum are strongly equivalent to well-guarded agents, we get by induction

\[ T(P\{F/W\}) = T(\sum_{i=1}^{n} (\alpha_i, P_i)\{F/W\}) = \sum_{i=1}^{n} T((\alpha_i, P_i)\{F/W\}) \leq (w)(\text{spawn}_w(F) \mid \sum_{i=1}^{n} \beta_i, Q_i) = (w)(\text{spawn}_w(F) \mid T(\sum_{i=1}^{n} \alpha_i, P_i)) = (w)(\text{spawn}_w(F) \mid T(P)) \] as desired.

**P** \(= [x = y]P'\). Then
\[ T(P\{F/W\}) = T([x = y]P'\{F/W\}) = [x = y]T(P'\{F/W\}) \leq [x = y](w)(\text{spawn}_w(F) \mid T(P')) \] (by induction) \(\sim\)
\[ (w)(\text{spawn}_w(F) \mid [x = y]T(P')) = (w)(\text{spawn}_w(F) \mid T([x = y]P')) = (w)(\text{spawn}_w(F) \mid T(P)). \]

**P** \(= (x)P'\). Then
\[ T(P\{F/W\}) = T((x)P'\{F/W\}) = (x)T(P'\{F/W\}) \leq (x)(w)(\text{spawn}_w(F) \mid T(P')) \] (by induction) \(\sim\) \( (w)(\text{spawn}_w(F) \mid (x)T(P')) \)
(assuming \(x \notin \text{n}(\text{spawn}_w(F)) \))
\[ (w)(\text{spawn}_w(F) \mid T((x)P')) = (w)(\text{spawn}_w(F) \mid T(P)). \]

**P** \(= !P\). Then
\[ T(P\{F/W\}) = T(!P\{F/W\}) = !T(P'\{F/W\}) \leq !\text{!}(w)(\text{spawn}_w(F) \mid T(P')) \] (by induction)
\(\sim\) \( !\text{!} \cdot (w)(\text{spawn}_w(F) \mid T(P')) \) (by lemma 3.3(iii))
\[ = (w)(\text{spawn}_w(F) \mid !T(P)) = (w)(\text{spawn}_w(F) \mid T(P)). \]

**P** \(= P' \setminus z\). Then
\[ T(P\{F/W\}) = T(P'\{F/W\}\setminus z) = T(P'\{F/W\})\setminus z \leq (w)(\text{spawn}_w(F) \mid T(P')\setminus z) \approx_{\text{!}} (w)(\text{spawn}_w(F) \mid T(P')) \]
(since \(z\) cannot be the subject of any synchronization of \(\text{spawn}_w(F)\)) \(\sim\)
\[ (w)(\text{spawn}_w(F) \mid T(P'\setminus z)) = (w)(\text{spawn}_w(F) \mid T(P)). \]

**P** \(= X(y)\).
Assume first \(X \neq W\). Then
\[ P\{F/W\} = X\{F/W\}\{y\} = X(y) = P \]
and
\[ (w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(X(y))) \sim \]
\[ (w)(\text{spawn}_w(F)) \] (since \(w \notin \text{n}(T(X(y)))) \sim \]
\[ 0 \mid T(P) \sim T(P) = T(P\{F/W\}). \]
Assume now \(X = W\). Then
\[ P(F/W) = W(y)\{F/W\} = F(y). \]

Hence
\[
(w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(W(y))) \geq T(F(y)) \quad \text{(by lemma 3.9(i))} = T(P).
\]

\[ \bullet P = X(X'). \]

We may distinguish 3 alternatives:

(i) \( X \not= W, X' \not= W. \) Then
\[ P\{F/W\} = X(X') = P. \]

Hence
\[
(w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(X(X'))) = (w)(\text{spawn}_w(F)) \mid T(X(X')) \quad \text{(since } w \not\in \text{fn}(T(X(X')))) \sim 0 \mid T(X(X')) = T(P) = T(P\{F/W\}).
\]

(ii) \( X \not= W, X' = W. \) Then
\[ P\{F/W\} = X(X')\{F/W\} = X(F). \]

Hence
\[
(w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(W(X'))) \geq T(F(X')) \quad \text{(by lemma 3.9(ii))} = T(P\{F/W\}).
\]

(iii) \( X = W. \) Then \( X' \not= W \) if \( P \) is well-sorted, and
\[ P\{F/W\} = X(X')\{F/W\} = F(X'). \]

Hence
\[
(w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(W(X'))) \geq T(F(X')) \quad \text{(by lemma 3.9(ii))} = T(P\{F/W\}).
\]

\[ \bullet P = X(G'), \text{ } G \text{ not a process variable.} \]

Assume \( X \not= W. \) Then
\[ P\{F/W\} = X(G\{F/W\}). \]

Hence
\[
(w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(X(G))) = (w)(\text{spawn}_w(F), \bar{u}, \bar{w}' \mid \text{rec}(\bar{u}' \mid \text{spawn}_w(G) )) \sim \bar{u}, \bar{w}' \mid \text{rec}(\bar{u}' \mid \text{spawn}_w(G)) \quad \text{(by expansion)} \\
\sim \bar{u}, \bar{w}' \mid \text{rec}(\bar{u}' \mid \text{spawn}_w(G\{F/W\})).
\]

(4) \text{Lemma 3.8) ~}
\[
(w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(W(G \{F/W\}))) \quad \text{(since } w \not\in \text{fn}(\text{rec}(\bar{u}' \mid \text{spawn}_w(G\{F/W\})))) \sim \\
\bar{u}, \bar{w}' \mid \text{rec}(\bar{u}' \mid \text{spawn}_w(G\{F/W\})).
\]

Assume \( X = W. \) Then
\[ P\{F/W\} = W(G\{F/W\}) = F(G). \]

since \( W \) does not occur in \( G \) if \( P \) is well-sorted.

Now
\[
(w)(\text{spawn}_w(F) \mid T(P)) = (w)(\text{spawn}_w(F) \mid T(W(G))) \geq T(F(G)) \quad \text{(by lemma 3.9(ii))} = T(P\{F/W\}).
\]

\textbf{Lemma 3.11} If \( S \) is a weak barbed bisimulation up-to \( \leq \), then \( S \subseteq \approx \).

\textbf{Proof:} By diagram chasing.
Lemma 3.12 $S = \{(P, T(P)) : P \in \Pi B\}$ is a weak barbed bisimulation up-to $\preceq$.

Corollary: For any $P \in \Pi B$, $P \preceq T(P)$.

Proof of the Lemma:

We will show the following:

1. Whenever $P \rightarrow P'$, then $P''$ exists s.t. $T(P) \Rightarrow P''$, and $P' ST(P') \preceq P''$.
2. Whenever $T(P) \rightarrow P''$, then $P'$ exists s.t. $P \rightarrow P'$ and $P' ST(P') \preceq P''$.

The proof is by induction on depth of inference. The cases that are possible are SUM, PAR, COM, MATCH, REP, RES and BLOCK.

**SUM:** $P = P_1 + P_2 \rightarrow P'$, $P_1 \rightarrow P'_1$.

By induction $T(P_1) \Rightarrow P' \sim T(P')$. Then

$T(P) \equiv T(P_1 + P_2) = T(P_1) + T(P_2) \Rightarrow P'' \sim T(P')$

On the other hand, if $T(P) \rightarrow P''$ by the SUM rule, then necessarily

$P \equiv P_1 + P_2$ and thus

$T(P) \equiv T(P_1 + P_2) = T(P_1) + T(P_2) \rightarrow P'' \sim T(P')$. By induction $P_1 \rightarrow P'_1$ for some $P'$ such that $T(P)' \sim P''$. But then $P \equiv P_1 + P_2 \rightarrow P'$ and $T(P') \preceq P''$ as desired.

For PAR, MATCH, RES, REP and BLOCK the proof is similar, since in all these case $T$ is a homomorphism.

**COM:** Suppose $P \xrightarrow{\tau} P'$ by COM.

Then $P = P_1 \parallel P_2$ for some agents $P_1, P_2 \in \Pi$. Two alternatives are possible now:

(i) $P' = (z)(P'_1 \parallel P'_2)$ for some name $z$, where $(z)$ may be void, and some agents $P'_1, P'_2 \in \Pi$ such that

$P_1 \parallel P_2 \xrightarrow{\tau} P'' \equiv (z)(P'''_1 \parallel P'''_2)$, and

$P_1 \xrightarrow{(z)F} P'_1, P_2 \xrightarrow{x(z)} P'_2$

for some name $x$. Thus, by lemma 8

$T(P_1) \xrightarrow{(z)F} T(P'_1), T(P_2) \xrightarrow{x(z)} T(P'_2)$.

Hence, by COM

$T(P) = T(P_1 \parallel P_2) = T(P_1) \parallel T(P_2) \xrightarrow{\tau} P'' \equiv (z)(T(P'_1) \parallel T(P'_2)) = (z)T(P'_1 \parallel P'_2) = T((z)(P'_1 \parallel P'_2)) = T(P')$.

(ii) $P' = (\tilde{y})(P'_1 \parallel P'_2)$ for some name vector $\tilde{y}$ such that $\tilde{y} \cap \text{fn}(P_1) = \emptyset$, and some agents $P'_1, P'_2 \in \Pi$ such that

$P_1 \parallel P_2 \xrightarrow{(\tilde{y})F} (\tilde{y})(P'_1 \parallel P'_2)$ and

$P_1 \xrightarrow{(y)F} P'_1, P_2 \xrightarrow{x(F)} P'_2$

for some abstraction $F$ and name $x$.

By lemma 9(i) and 9(ii), we have

$T(P_1) \xrightarrow{(\tilde{y})F} (\tilde{y})(T(P'_1) \parallel \text{spawn}_w(F)), w \not\in \text{fn}(P'_1)$,

$T(P_2) \xrightarrow{x(F)} T(P'_2), P'_2 \{F/W\} = P'_2$.

Hence, by COM, and since

$w \not\in \text{fn}(P'_1) \Rightarrow w \not\in \text{fn}(T(P'_1)), \tilde{y} \cap \text{fn}(P_2) = \emptyset \Rightarrow \tilde{y} \cap \text{fn}(T(P_2)) = \emptyset$, 254
then
\[ \mathcal{T}(P) = \mathcal{T}(P_1 | P_2) = \mathcal{T}(P_1) \cup \mathcal{T}(P_2) \overset{r}{\rightarrow} P'' \equiv \\
(\langle u \rangle)(\langle \bar{y} \rangle(\mathcal{T}(P_1') | \text{spawn}_u(F) | \mathcal{T}(P_2'))) \sim \\
(\langle \bar{y} \rangle(\mathcal{T}(P_1') | \mathcal{T}(P_2') \{F/W\})) \ (\text{by lemma } 3.10) \sim \\
(\langle \bar{y} \rangle(\mathcal{T}(P_1') | \mathcal{T}(P_2')) = (\langle \bar{y} \rangle(\mathcal{T}(P_1') | P_2') = \\
(\langle \bar{y} \rangle(\mathcal{T}(P_1') | \mathcal{T}(P_2')) = \mathcal{T}(P') \text{ as desired.})

Suppose now \( \mathcal{T}(P) \overset{r}{\rightarrow} P'' \) by COM.
Then \( P = P_1 | P_2 \) for some agents \( P_1, P_2 \in \Pi \), and again we have two alternatives:

(i) \( P'' = (z)(Q_1 | Q_2) \) for some agents \( Q_1, Q_2 \in \pi \), and
some name \( x \), where \( (z) \) may be void, such that for some first-order name \( x \):
\[ \mathcal{T}(P_1) \overset{(z)x}{\rightarrow} Q_1, \mathcal{T}(P_2) \overset{x}{\rightarrow} Q_2.
Then, by lemma 3.6,
\[ P_1 \overset{(z)x}{\rightarrow} P_1', P_2 \overset{x}{\rightarrow} P_2' \]
for some agents \( P_1', P_2' \in \Pi \) such that
\[ \mathcal{T}(P_1') = Q_1, \mathcal{T}(P_2') = Q_2 \).
Hence, by COM
\[ P = P_1 | P_2 \overset{r}{\rightarrow} P' \equiv (z)(P_1' | P_2') \]
and
\[ \mathcal{T}(P') \equiv \mathcal{T}((z)(P_1' | P_2')) = (z)(P_1' | P_2') = (z)(\mathcal{T}(P_1') | \mathcal{T}(P_2')) = \\
(z)(Q_1 | Q_2) = P'' \text{ as desired.}
(ii) \( P'' = (w)(Q_1 | Q_2) \) for some agents \( Q_1, Q_2 \in \pi \)
such that for some name \( x \)
\[ \mathcal{T}(P_1) \overset{(w)x}{\rightarrow} Q_1, \mathcal{T}(P_2) \overset{x}{\rightarrow} Q_2.
Then, by lemma 3.7,
\[ P_1 \overset{(w)x}{\rightarrow} P_1', P_2 \overset{x}{\rightarrow} P_2' \{F/W\}, \]
for some name vector \( \bar{y} \), abstraction \( F \), and agents \( P_1', P_2' \in \Pi \) such that
\[ (\langle \bar{y} \rangle(\mathcal{T}(P_1') | \text{spawn}_u(F)) \sim Q_1, \mathcal{T}(P_2') = Q_2 \).
Hence, by COM, assuming \( \bar{y} \cap \text{fn}(P_2) = \emptyset \),
\[ P = P_1 | P_2 \overset{r}{\rightarrow} P' \equiv (\langle \bar{y} \rangle(P_1' | P_2' \{F/W\}) \]
and
\[ \mathcal{T}(P') = \mathcal{T}((\langle \bar{y} \rangle(\mathcal{T}(P_1') | P_2' \{F/W\})) = (\langle \bar{y} \rangle(\mathcal{T}(P_1') | \mathcal{T}(P_2') \{F/W\}) \sim \\
(\langle \bar{y} \rangle(\mathcal{T}(P_1') | (w)(\text{spawn}_u(F) | \mathcal{T}(P_2'))) \ (\text{by lemma } 12) \sim \\
(\langle \bar{y} \rangle(\mathcal{T}(P_1') | \text{spawn}_u(F)) \sim \mathcal{T}(P_2')) \ (\text{assuming } w \notin \text{fn}(\mathcal{T}(P_1'))) \sim \\
(\langle \bar{y} \rangle(\mathcal{T}(P_1') | \text{spawn}_u(F)) \sim \mathcal{T}(P_2')) \ (\text{since } \bar{y} \cap \text{fn}(P_2') = \emptyset \Rightarrow \bar{y} \cap \text{fn}(\mathcal{T}(P_2')) = \emptyset) \sim \\
\sim \overset{(w)(Q_1 | Q_2) = P'' \text{ as desired.}}{=}
\]

**Theorem 3.1**: \( P \approx Q \iff \mathcal{T}(P) \approx_r \mathcal{T}(Q) \), i.e. \( \mathcal{T} \) is both sound and complete with respect to weak barbed equivalence in \( \Pi B \) and weak reduced equivalence in \( \pi B \).

**Proof**:

**Soundness**: Assume \( \mathcal{T}(C)[\mathcal{T}(P)] \approx \mathcal{T}(C)[\mathcal{T}(Q)] \) for every static context \( C[\cdot] \in \Pi B \), where \( P \) and \( Q \) are agents in \( \Pi B \). By Lemma 3.1, this implies that
\[ \mathcal{T}(C[P]) \approx \mathcal{T}(C[Q]). \]
Then by lemma 3.12 and transitivity of \( \approx_r \), \( C[P] \approx C[Q] \). Since this is true for every static context \( C \in \Pi B \), then \( P \approx_r Q \).

**Completeness**: If \( P \approx Q \), where \( P \) and \( Q \) are agents in \( \Pi B \), then \( C[P] \approx C[Q] \) for all static contexts \( C \in \Pi B \). Then by the lemma and transitivity of \( \approx_r \), \( \mathcal{T}(C[P]) \approx \mathcal{T}(C[Q]) \). By Lemma 3.1 we get \( \mathcal{T}(C)[\mathcal{T}(P)] \approx \mathcal{T}(C)[\mathcal{T}(Q)] \), and thus
the transformation is complete with regard to those agents and contexts in the rage of the transformation $\mathcal{T}$. 