Abstract

We investigate the possibility to design a simple concurrent object-oriented language with semantics defined in terms of a mobile calculus. First, we review in short some of the high-level concurrent features used in other languages. Then we describe basics of the $\pi$-calculus and the PICT language. The core of the PICT language is just a small extension of the $\pi$-calculus. These two languages will serve as our semantic domains.

The language we design – School$^{98}$ is based on simple Smalltalk-like object model extended with concurrent semantics. The language includes only four program control structures – message passing, new object creation, sequential composition and assignment to instance variables. The main concurrent features of the language are asynchronous message passing, data-driven synchronisation and active objects. We try to keep the data-flow as explicit as possible. The semantics is built in systematic structural fashion, creating one or two translation functions per syntactic category. We keep the translation local for each syntactic category.

The second part of the thesis exploits the possibility to add type system to our language. We extend the language with the object types proposed in [PT94]. The addition of the type system requires changes in the language design. We are forced to extend the set of language primitives with control structures (if-then-else, while-do, etc.), that were previously implemented as methods. Furthermore, the translation functions cannot be as local as in the untyped language. The typed version of the language is translated to the core of the PICT language. Its type system is ideally suited for the typed objects designed according to [PT94]. The translation to the PICT language opens the possibility for an easy implementation.
Acknowledgements

In first place, I would like to thank Damas Gruska, my supervisor, because without his invaluable guidance and encouragement this thesis would be just a piece of paper with a bunch of meaningless word. I also would like to thank Augustín Mrázik for access to his library, where many of the cited papers come from. I am sorry for disappointing him, but this thesis was all-important for me.

A special thanks goes to colleagues Gabriela Polčíková and Martina Rohál’ová, who supported me with various course-notes, and all the members of the artificial intelligence group at the Comenius University, Bratislava. A very special thanks goes to Alena Maarová for helping me with my English.

Last but not least, I thank my parents for great support through out my university study and especially during the last three months, during which most of this thesis was created.
Declaration

This diploma thesis has been composed by myself. The ideas and results contained in it, unless otherwise stated, are my own.
Contents

Chapter 1 Introduction 7
  1.1 COOP languages................................................................. 8
  1.2 The $\pi$-calculus ............................................................. 8
  1.3 Language PICT .................................................................. 8
  1.4 Language School/98 ........................................................... 9
  1.5 Language School/98-T ....................................................... 9
  1.6 Notation............................................................................. 10

Chapter 2 COOP languages 11
  2.1 Library.................................................................................. 12
     2.1.1 Karaormann & Bruno..................................................... 12
     2.1.2 Kobetič & Neurath...................................................... 12
  2.2 Language extension ................................................................ 13
     2.2.1 Eiffel extension ............................................................ 13
     2.2.2 CEiffel......................................................................... 15
     2.2.3 Eiffel// ...................................................................... 17
  2.3 New language........................................................................ 18
     2.3.1 Actor languages........................................................... 18
     2.3.2 ABCL/1.................................................................... 20
     2.3.3 POOL ....................................................................... 22
  2.4 Summary............................................................................. 24

Chapter 3 The $\pi$-calculus 26
  3.1 The monadic $\pi$-calculus...................................................... 26
     3.1.1 Syntax....................................................................... 27
Chapter 1

Introduction

“The behaviour of computer systems, human information processing systems, corporate organisations, scientific societies, etc. are the result of highly concurrent (independent, co-operative, or contentious) activities of their components.” For modelling and studying such systems, “it is necessary to develop an adequate formalism in which various interactions of objects can be expressed naturally.” These two citations are taken from [YST+87] and they are eleven years old. Despite their age, these citations are still topical, because there is no programming system that can provide satisfactory mechanisms to model all the objects we would like to.

The today’s world of Internet and networking needs systems that use the available parallelism, concurrency and distribution abilities. It is important to design simple transparent and flexible programming environments that can take advantage of all the available processing power. Concurrent systems are complex and therefore it is required to have very robust and provably correct system that leads programmer to good solutions. The concurrent programming system should be accompanied with a proof system that allows uncomplicated property checking.

We try to contribute to the discussion about the appropriate programming environment with our proposal. We synthesise well-known ideas into a new simple and flexible aggregate. The main idea of this thesis i.e. to give semantic interpretation of an object-oriented language in terms of a mobile calculus can be traced back to 1990. At that time, David Walker presented in his paper [Wal90] the potential of the $\pi$-calculus as a semantic domain.
1.1 COOP languages

We begin this thesis with a short review of some of the approaches to creation of a concurrent object-oriented environment. We present a short list of papers and sketch the proposed solutions. This review is by no means complete, but it shows some of the desired basic features. The review concentrates on high-level features that are present in these solutions and explains their meaning in each case. Some of the features are present in every solution (data-driven synchronisation) and we focus right on these features, because they seem to form the basic set of features that should be present in every solution. Our proposed language is then build from the subset of the shown features. We create a simple language that includes only the most elementary features (asynchronous message passing, new object creation and sequential composition) and we leave the specification open for possible future extensions.

1.2 The $\pi$-calculus

There are many possible approaches to define semantics of a language. There are two standard semantic approaches – operational and denotational. The semantics in terms of a mobile calculus, particularly the $\pi$-calculus, is an approach that can fit into the category of structural operational semantics. We describe in Chapter 3 the basic features of the monadic $\pi$-calculus and the polyadic $\pi$-calculus. The monadic $\pi$-calculus is elegant in the presentation of the core features. The polyadic $\pi$-calculus offers more flexibility in applications. We also describe the encoding of basic data structures that we use later. The main advantage of the polyadic $\pi$-calculus is the possibility to build a type system for it. Therefore, we use the polyadic $\pi$-calculus as our basic semantic domain, despite the fact that the first part of the thesis deals with an untyped language. This approach allows us to reuse much of the translation of the untyped language in the case of the typed one.

1.3 Language PICT

As we said earlier, we try to synthesise the already known results into the new whole. Therefore, we decided to use the language PICT and its type system as a
basis for the translation of our language extended with typing. In addition, we use the typing of objects presented in [PT94]. This paper presented typing of object that is almost directly usable in the language PICT. It may seem that our focus slightly moved from mobile calculi to some programming language, but the language PICT is based on the polyadic $\pi$-calculus with a polymorphic type system. Its core part is almost the original polymorphic $\pi$-calculus. Every high-level construct of this language can be translated into the core language. The language PICT was build right for the purposes of experiments as its authors point out in [PT97]: “Rather than commit to a particular high-level object model in PICT, we chosen to provide a framework for experimenting with variety of designs.” We describe the most important features of this language in Chapter 4.

1.4 Language School/98

The main part of the thesis begins in Chapter 5. We design the language School/98 (Simple Concurrent High-level Object-Oriented Language). The goal is to design the language with minimal number of syntactic constructs and with precise semantics in terms of the $\pi$-calculus. Based on the Chapter 2, we define the required properties of our language. We equip the language School/98 with only the most basic syntactic constructs – message passing, new object creation, sequential composition and assignment to instance variables. The rest of the program control structure we define in terms of methods – special section 5.3 is devoted to built-in classes. The language specification is open for possible future enhancements. Through out the Chapter 5 we point out the possible encoding of many high-level constructs described in Chapter 2.

The proposed language concentrates on the core features and it would require further work to make it a full-fledged high-level language.

1.5 Language School/98-T

Untyped languages can offer big flexibility, but they also suffer from possible failures caused by ability to pass wrong data to some function. Therefore, in Chapter 6 we try to add a type system to our language. Despite the change of a semantic domain, we try to reuse as much of the first translation as possible. The addition of the type system brings some necessary changes in the language syn-
tax. Nevertheless, the programs written in the untyped language should be almost
the same as in the typed version of the language. In the typed language, we also
introduce a translation that is more complex. This translation will make the
syntax of the language closer to the syntax of standard object-oriented languages.
First part of the Chapter 6 requires familiarity with [PT94].

1.6 Notation

In the following text, we decided to use different notation then the standard π-
calculus notation. This notation was inspired by the PICT language and makes the
standard π-calculus code look similar to the code written in language PICT. The
table shows the new notation together with the standard π-calculus counterparts:

<table>
<thead>
<tr>
<th>π-calculus</th>
<th>our notation</th>
<th>PICT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{xy}.P$</td>
<td>$x!y \cdot P$</td>
<td>output prefix</td>
</tr>
<tr>
<td>$\overline{xy}.0$</td>
<td>$x!y$</td>
<td>asynchronous output</td>
</tr>
<tr>
<td>$x(y).P$</td>
<td>$x?y \cdot P$</td>
<td>$x?y = P$</td>
</tr>
<tr>
<td>$(\nu x)P$</td>
<td>$(\nu x)P$</td>
<td>(new $x \cdot P$)</td>
</tr>
<tr>
<td>$!P$</td>
<td>$*P$</td>
<td>replication</td>
</tr>
<tr>
<td>$!x(y).P$</td>
<td>$*x?y \cdot P$</td>
<td>$x?*y = P$</td>
</tr>
</tbody>
</table>

We also often omit the null agent as the last process e.g. we write $x?y$ instead of
$x?y.0$. 
Chapter 2

COOP languages

There are several approaches one can take in designing a programming environment that has both object-oriented and concurrent features. Each approach has its advantages and disadvantages. We show in this chapter three ways of building a concurrent object-oriented programming language. Each section starts with a list of advantages and disadvantages of each approach.

Section 2.1 shows a library approach i.e. building a concurrent object library on the top of an existing sequential language. The library frequently uses operating system services to achieve the concurrent operation.

Section 2.2 shows ways of bringing concurrency into an existing sequential object-oriented language. The usual ways include the change of semantics of the language, introduction of a new syntactic construct and possibly building a library tool for a flexible adaptation of the new concurrent features.

The last section, section 2.3, shows three examples of COOP languages built ‘from scratch’ i.e. this section presents three new languages.

Two features are common for every object-oriented system and they are preserved in almost every system presented in this chapter. The first is the ability of the system to change its topology. Objects, even the presented concurrent objects (active objects), can exchange their addresses and the whole system of objects then exhibits dynamic topology. The other property, taken from sequential languages, is that the order of messages is preserved. Unless it is stated explicitly, the order of messages will be preserved in every presented system.
2.1 Library

The library approach is easy to implement and quickly available to business applications, because the library is usually built on the top of a sequential language. It does not force the programmer to learn any new language constructs, it only offers new features in form of a library for the programmer. On the other side, the library approach cannot create as flexible and as transparent programming environment as some other approaches.

2.1.1 Karaormann & Bruno

Murat Karaormann and John Bruno show in their paper [KB93] a design of a class library. The design is based on a class `CONCURRENCY`. This class includes a method `split` that explicitly starts a new active object. Messages sent to the active object must preserve special protocol i.e. a method `remoteInvoke`. Using this method, the messages are passed `asynchronously`. The method `remoteInvoke` returns a unique request number. The result of such a method call can be accessed through the `claimResult` method and the request number. The call to `claimResult` blocks until the result is available. The blocking access to the result of a message call is a form of `data-driven synchronisation`.

The passive objects cannot be shared, because when a passive object is sent to a remote active object it is copied. The remote object receives a deep copy of every passive object.

The system presented by Karaormann and Bruno can be seen as a way of placing the Eiffel language on the top of the concurrency features that exist in the underlying processor or operating system. It is different then building-in the concurrency into a language.

2.1.2 Kobetič & Neurath

Two diploma theses of Kobetič and Neurath show nicely the advantages and disadvantages of the library approach. These theses also show the implementation of such library in Smalltalk-80.

The relevant concurrent features studied in the diploma thesis of M. Kobetič [Kob95] are `synchronous` and `asynchronous message passing` together with `data-driven synchronisation`. The library allows distribution of objects across several
Smalltalk images that can reside on different computers connected over a network. Message passing protocol is the same as in Smalltalk-80 i.e. the programmer does not have to know the exact location of the receiver. This distribution transparency is achieved by using an object request broker built according to the CORBA standard. Asynchronous message passing is complemented with data-driven synchronisation, but the synchronisation procedure is explicit. The other non-transparency in the proposed solution is the start-up if the whole system.

The diploma thesis of P. Neurath [Neu95] deals with transaction management in distributed object environment. The mechanisms investigated in this thesis are very powerful and in some ways go beyond the standard constructs (e.g. object locking for an exclusive access) used in the other solutions. On the other side, some constructs (e.g. conditional locking) are not supported.

2.2 Language extension

Language extension usually consists of adding new keywords, modifiers and/or change of semantics and a new supplementary library tools. This median way does avoid some disadvantages while preserving the advantages of the library approach e.g. the transparency of the programming environment is higher while the sequential language remains the same. However, it cannot provide as clean solution as building a new language. In this section, we survey some extensions to the language Eiffel. We concentrate on the essential aspects of these extensions not the technicalities that are necessary for the implementation in the Eiffel language.

2.2.1 Eiffel extension

The author of the language Eiffel proposed in [Mey93] an extension that introduces some concurrent features to the language. This solution is constrained by the fact that the author wanted to reuse sequential code as much as possible and he wanted to make a minimal change in the language design. The main concurrent features explained in turn are:

- separate objects
- process objects
- object locking with constraints
CHAPTER 2. COOP LANGUAGES

- *asynchronous message passing*
- *lazy-wait* (data-driven synchronisation)

Any object can be created with a modifier `separate` – this keyword is the only syntax change. Keyword `separate` can also modify a class – every instance of this class will be created as a separate object. Such separate object is assigned to a virtual processor. Virtual processor is an entity with processing power and it can execute the separate object. Virtual processors can reside on one computer (sequential or parallel) or they can be distributed over a network. The language design does not specify the distribution mechanism and leaves it to the underlying operating system. The distribution can possibly be controlled through a special library that uses operating system services.

Active objects are not supported directly by the syntax, because the author thinks that it would be too restrictive. He proposes a different way. Active objects are instances of the descendants of the class `PROCESS`. Class `PROCESS` includes one special method – `live` – that models the body of the active object.

The proposed language extension includes implicit locking. When a method of an object is called and it includes a separate object as a parameter (separate parameter) the object is locked for an exclusive access. The method is blocked until the lock is acquired. The exclusive access is guaranteed throughout the execution of the method. If a precondition of the called method includes a condition on the separate parameter then the lock of the object is acquired only when the condition is satisfied, otherwise it must wait for the condition to become satisfied. The locking also introduces a synchronisation mechanism, because a method can call a separate object only if it owns an exclusive access to it! This means that the separate object must be in sort of idle state. When a method owns an exclusive access to a separate object, it can send messages to this object asynchronously.

Other synchronisation mechanism available is the ability of an object to execute only one method at a time. Even express messages are treated in a way that they interrupt the current method causing an exception in the client. A life of an object is "a sequence of transitions between consistent states". This allows better proofs of correctness.
The language extension also provides an implicit data-driven synchronisation, called by the author a lazy-wait policy. When an asynchronous call was made, the caller continues its execution until the moment when it accesses the variable containing the result returned by the call. When the result is not available, yet, the caller is blocked until the arrival of the actual result. When the result becomes available, the execution continues.

2.2.2 CEiffel

Peter Löhr in [Loh93] uses interesting technique to enhance the sequential language Eiffel with some concurrent features, while preserving the ability to compile the concurrent code using a sequential compiler. He adds special modifiers that are in form of comments for the sequential compiler, but the concurrent compiler recognises them as the concurrent modifiers. The main concurrent features of his extension are:

- controlled objects
- user-controlled compatibility
- delayed acceptance
- autonomous object (active objects)
- asynchronous message passing
- lazy synchronisation (data-driven synchronisation)

As in the previous subsection where concurrent object were those specified as separate, here the concurrent objects are annotated with the control annotation \texttt{-!-}. The concurrent objects act in parallel. The semantics does not specify the actual handling of these active objects i.e. it does not specify the actual amount of parallelism used during the execution.

The default behaviour of a controlled object is atomic i.e. only one method is active at a time. The programmer can relax this strict property. The programmer can specify which methods are compatible – in the sense that they can be executed concurrently. When a method definition is annotated with the compatibility annotation \texttt{-|| operation list --} it means that it can be executed concurrently with the methods listed in the operation list. The compatibility relation is symmetric, but it need not be transitive or reflexive – transitivity and reflexivity must be stated explicitly. If the operation list is empty the method is compatible with itself and all the methods that are annotated with this annotation – this is
usually used for read-only methods. The semantics specifies the mutual exclusion of incompatible methods – when an incompatible request arrives, it remains pending until all the current activities terminate. After the termination of these activities several requests may start depending on their compatibility and the order of their arrival – the order of messages is preserved. Deferred methods must not be annotated with the compatibility annotation. In addition, methods that call deferred methods should not be annotated this way (the implementers of the deferred method could not fulfil this obligation).

Standard preconditions are divided into two parts divided by the delay annotation \(--\theta--\). The checker part is a standard Eiffel precondition and the \textit{guard} part works as follows: the method is blocked until all the conditions in the guard are satisfied. The conditions are checked every time an activity in the object terminates. The delay annotation can also be used in postconditions. The delay in postconditions can serve for synchronisation purposes.

Active objects are supported in a form of autonomous objects. An autonomous object includes special method or methods that are annotated with the autonomy annotation \(--\rightarrow--\). These methods must have zero arity and an empty checker. When an autonomous object is created, its autonomous methods are started automatically. After an autonomous method terminates, it is implicitly restarted! Autonomous methods, controlled by an appropriate guard, can be used to do the post-processing after some method.

By default, all the messages are passed synchronously. The programmer must explicitly annotate those methods that can be invoked asynchronously. Only a method annotated with the asynchrony annotation \(--\textbf{v}--\) can be invoked asynchronously. Asynchrony is sometimes referred to as the vertical concurrency (asynchrony annotation includes \textbf{v}) and autonomy as the horizontal concurrency (autonomy annotation include \textbf{>}). This comes from the vertical nature of invocation in a functional hierarchy.

Together with asynchronous message passing, CEiffel also supports data-driven synchronisation called lazy synchronisation. This synchronisation technique is the same as lazy-wait described in the previous subsection. An activity is blocked only when it accesses the actual result of a previous asynchronous call.
and it is still not available. The activity remains blocked until the actual result arrives.

CEiffel does not include any construct for locking an object for the exclusive use.

2.2.3 Eiffel//

In the language Eiffel//, pronounced as Eiffel parallel, the extension is achieved only at the level of semantics and a library tool. The author, in his paper [Car93], shows his concurrent extensions of the Eiffel language in the light of the concurrent object-oriented software design, but the software design issues are unimportant in this thesis. We concentrate on the concurrent features of Eiffel//:

- process objects
- no sharing of non-process objects
- synchronous message passing with asynchronous service
- wait-by-necessity (data-driven synchronisation)

Process is an object executing the prescribed behaviour defined by its class inheriting from a class PROCESS. The class PROCESS includes a message live that serves as a default process body. Process object executes its live method automatically after its creation and when this method terminates, the process terminates.

Non-process objects are private for each process object. They are not shared! This requirement serves as a guarantee against future changes in a concurrent system – concurrent systems are complex and a small change can have enormous side affects. To avoid sharing in message passing, the non-process parameters of a communication are passed by deep copy, not by reference. Process parameters are passed by reference.

Messages are transmitted synchronously and a handshake protocol is employed. After the receiving object acknowledged a request, the sending object continues its execution. The receiving object does not serve the message right away. The received message is put to the internal list structure and it is up to the method live to choose which received message is next to be served. The default method live serves the messages in the order of their arrival. A redefinition of this routine can modify this default behaviour. The class PROCESS includes several methods that are flexible and robust enough to allow various request-serving
policies to be implemented. For example, process can protect itself against a new message reception by signalling `ignore` and then remove this protection by signalling `continue`. Despite the fact that the actual message passing is synchronous, the whole mechanism of request serving is asynchronous.

Synchronisation can be achieved by wait-by-necessity principle. This is again the same synchronisation system as was the lazy-wait and lazy synchronisation. The new capability in the Eiffel// is that the synchronisation aside being implicit can be programmed explicitly through methods `wait` and `awaited`. This is the only synchronisation mechanism available.

The implementation of Eiffel// is distributed and it assigns one OS process per process object. The communication is achieved using sockets.

### 2.3 New language

Creating a new language offers freedom in choice of language constructs and it allows many semantic approaches. Building a new language allows a designer to experiment with the language. The major drawback is the low applicability of such a solution. Nevertheless, new language can provide clean and transparent solution for the programmer.

#### 2.3.1 Actor languages

Actor languages are among the earliest languages that tried to exploit high-level concurrent object-oriented language features. Actor languages combine some of the attractive features of functional and object-oriented programming.

Functional programming is inherently concurrent in the sense that it allows the possibility of concurrent execution of all sub-expressions in a program. The main reason for this property is explicit data-flow, and at the time the actor languages were created there was not known a pure functional solution for the problem of shared objects with internal state. Pure functional solution to this problem was later given by using monads – see [Wad95], and fudgets – see [HC95], and fudgets can be in many points compared with actors.

Object-oriented programming offers an unrivalled modularity, extensibility and compositionality. The authors of actor languages in particular Agha and Hewitt in [AH87] wanted to design an architecture that supports large-scale concurrent
open systems and these properties of object-oriented systems were among the most important in designing such an architecture. Actor languages usually do not include inheritance.

Actor languages have these concurrent properties:
- asynchronous message passing
- futures (data-driven synchronisation)
- active objects (actors)

An actor system consists of a number of actors each one with its own unique mail address and a mail system that is responsible for message delivery. The mail system buffers the massages for each actor in its mailbox. However, the important difference to other concurrent systems is that the message delivery is nondeterministic! The actor then picks the messages one by one in the order of their arrival. The mail address is used as a reference to the actor. The actor can send messages to those actors whose mail address it knows; each actor knows its own address by default. The actor with its mailbox can be depicted as:

The ability to change the local state while preserving referential transparency is achieved by specifying the replacement behaviour. The replacement actor will then accept the next communication received at the mail address. Replacement is achieved using become command. This replacement behaviour is shown in the following picture, where A is the original actor and A’ is the replacement actor:

The actions of the replacement actor can be carried out concurrently with other actions performed by the replaced actor – the replacement process is intrinsically
concurrent. This property allows system speed-up by pipelining the actions to be performed. As soon as the replacement actor is created, it can start to accept communication despite the fact that the replaced actor is still working. This means that the only speed limitations in the actor system come from logical dependencies in the computation and the limitations imposed by the hardware. Whereas in von Neumann architectures e.g. imperative languages, the data dependencies caused by assignments to a global store restrict the degree of pipelining that can be realised. These data dependencies also make the determination of possible concurrent statements harder; the data-flow analysis is needed.

Each received request is accompanied by a mail address of the customer to which a reply is to be sent. It also contains a mail address to which a complaint can be sent (error handler). Authors ([AH87]) think that it is useful to allow error handling to be separated from the successfully completed transactions.

As an example of the actor language, we show some of the features of the language Act3 presented in [AH87]. The Act3 language is written in Lisp-based language Scripter. The language is based on a simple actor language Act. All the high-level structures of Act3 can be translated into the core language Act. The commands in the code of an actor can be executed concurrently. This differentiates the Act3 from systems based on communicating sequential processes, where commands must be executed sequentially. For example the expression:

\[
(* \ (\text{call} \ \text{RangeProduct} \ (\text{with} \ \text{low} \ \text{lo}) \ (\text{with} \ \text{high} \ \text{hi}))
\ (\text{call} \ \text{RangeProduct} \ (\text{with} \ \text{low} \ \text{mid}+1) \ (\text{with} \ \text{high} \ \text{hi})))
\]

creates two actors which compute concurrently and then their results are multiplied. This behaviour is called unserialised. Act3 allows lazy and eager evaluation strategies for expressions.

2.3.2 ABCL/1

Language ABCL/1 described in [YST+87] has its roots in actor languages, but extends the basic actor architecture in many points. The main difference is the message reception. In actor languages, an actor always serves the first message in the mailbox. In ABCL/1, authors introduce two types of messages – ordinary and express. Each object has two mailboxes each for one type of messages. The service of express messages has higher priority than the service of ordinary messages i.e. when an express message arrives to the express mailbox an object
interrupts its ordinary activity and promptly serves the express message. The express message can specify whether upon its termination the previous activity is resumed or aborted. The object definition specifies the type of message (express, ordinary). The object can also use the (atomic ...) block to protect enclosed statements from interruption by an express message.

To avoid difficulties possible in the original actor systems new assumption concerning the order of messages was made in ABCL/1. The temporal ordering of the message transmissions from the sender (according to its internal clock) is preserved in the temporal ordering of the message arrivals in the receiver (according to its internal clock). This is a standard assumption used in almost all the concurrent systems presented in this chapter. The concurrent features in ABCL/1 are:

- active objects
- past type message passing (asynchronous)
- now type message passing (synchronous)
- future type message passing (asynchronous + data-driven synchronisation)
- explicit reply destination
- constrained message acceptance
- parallel construct

Objects in ABCL/1 are active just as actors. Objects that have internal state – called serialised – can execute only one message at a time, while objects that have no internal state – called unserialised – can handle more than one message in parallel.

Objects can be in one of three states. The first two are standard, known from other languages, but the third one is new. The three states are:

- dormant – initial; no activity
- active – executing an action, message
- waiting – waiting for a special message

The select command puts an object into the waiting state. After the select command, the programmer specifies the messages that can be accepted in this state together with a new message body for each message. In waiting state, the object searches for the first message that can be possibly accepted in its mailboxes.
The *past type* message passing ([0<=M] – ordinary, [0<==M] – express) is a standard form of asynchronous message passing – the sender continues its execution independently. The *now type* message passing ([0<==M], [0<==M]) acts like a normal synchronous message passing – the sender waits for the result. The *future type* message passing ([0<=M$X$], [0<==M$X$]) is a form of asynchronous message passing together with data-driven synchronisation. The special variable x is a queue. The sender can explicitly check the availability of results. The queue can transmit several results.

The name of the sender and the reply destination are sent as parameters of every message. The reply destination can be explicitly specified only in the past type message passing. The rest of the message passing types specify the reply destination and include it implicitly. This means that in the past type message passing the programmer can specify where the result is sent e.g. the result can be received by a different object then the sender. On the receiver’s side, the reply destination can be used instead of a return statement. For example, the receiver can send the result to the reply destination and continue its execution.

A message body can contain a constraint that specifies when the message can be accepted. For example, this constraint can examine the sender. When the constraint is not satisfied the message is discarded.

The parallel construct \{ msg_pass. ... msg_pass.\} sends all the messages in parallel. The execution of this statement ends when all the components terminate their execution. The parallel construct together with now type message passing can be used for synchronisation. Special type of this parallel construct is multicast – one message is sent to a list of objects.

### 2.3.3 POOL

POOL (Parallel Object-Oriented Language) described in [Ame87] and [Ame90] is a pure object oriented language with active objects and inheritance. The main concurrent features are:

- active objects
- synchronous message passing
- explicit message acceptance
- method post-processing
Every object is active and contains a body. The body specifies the actions of an object. If the programmer does not specify any object body, the object will be assigned the default body. The default body just answers all the incoming requests – infinite loop of the \texttt{ANSWER} statement. Objects with default body can be thought of as passive objects.

Messages are sent synchronously and on the receiver’s side, they must be accepted explicitly. The answer statement \texttt{ANSWER(m_1,\ldots,m_n)} must be used to signal that an object is willing to serve messages \(m_1\) to \(m_n\). This explicit acceptance is called rendezvous and is similar to that of the Ada language. The language POOL also contains a more complicated \texttt{SELECT} statement that answers messages conditionally. To allow recursion the message passing within one object is executed as a standard function call.

The asynchronous service in the POOL language is achieved using method post-processing. The method definition has this syntax:

```plaintext
METHOD m(...) 
  ... 
  RETURN exp 
  POST 
  ... 
END m 
```

The method body is executed just as in standard object-oriented languages. However, after the return statement the programmer can specify a post-processing section. After the return statement, the sender is free to continue its execution and the post-processing section is started in parallel to the sender. The receiver executes only the post-processing section and after it terminates, the \texttt{ANSWER} statement terminates as well. Then the receiver can continue its execution with the statement following the \texttt{ANSWER} statement.

The POOL language also includes routines, which closely resemble class methods but they are not equivalent. For example, routines are used for the creation of new objects.
2.4 Summary

We conclude this chapter with the list of the features that come to question in the design of a concurrent object-oriented programming environment. The main questions can be summarised into four points:

- **type of message passing** (sync./async.)
- **active objects** (yes/no)
- **single-threaded of object** (yes/no)
- **synchronisation mechanisms** (data-driven synchronisation and/or ?)

Answers to these questions will indicate the amount of available concurrency and the amount of control the programmer has over the execution of the system. We think that some of the questions are in many cases answered in similar fashion. We point out three items that will guide our design of the language School\textsuperscript{98} and its extension School\textsuperscript{98-T}.

In some form, data-driven synchronisation was included in every presented system. This synchronisation mechanism seems to be very natural, because it controls the execution of a program according to data-flow. Coupled with this synchronisation mechanism is always some construct that generates asynchrony or parallel activity. The simplest one is asynchronous message passing and we therefore decided to include these two features in our language.

Object-oriented modelling offers big flexibility and it allows entities to be modelled very naturally. In the real world, many objects contain apparatus that allows them to change upon their will or by some other object. This apparatus usually works by itself. Therefore it appears essential to assign processing power to every object in our language – we unite the notion of object and process.

One more idea is common in almost all the presented solutions. Except the original actor model, every other system preserves the order of messages. The preservation of message order is an important assumption that can significantly simplify the programming. It is also useful in the modelling of the real world objects, because many things in real life are organised on the first-come-first-serve basis. Nevertheless, there are some situations in life, that must be handled with higher priority and therefore the programming environment should allow modifications of the basic message-service scheme.
These ideas lead directly to design decisions and they are quite understandable in the light of this chapter, but the rest is not that clear and we defer the complete description of the desired language features to Chapter 5.
Chapter 3

The $\pi$-calculus

In this chapter we describe one of the mobile calculi that was developed in 1989 by Milner, Parow and Walker. The first two papers [MPW89a] and [MPW89b] show the basic calculus, examples and basic properties. The first version of the $\pi$-calculus was monadic i.e. only one channel name could be sent along a channel. In section 3.1 we show some of the core features of monadic $\pi$-calculus. Most of these features also apply for the polyadic $\pi$-calculus. Section 3.1 is based on two papers [MPW89a] and [Mil91].

In 1991 Milner modified the $\pi$-calculus mainly to allow tuples to be sent along a channel. This change that looks like a small notational change brought some new aspects into play. Section 3.2 is devoted to these new characteristics of the polyadic $\pi$-calculus and is based on [Mil91]. The tutorial of the polyadic calculus can be found in [Mil91].

The last section of this chapter deals with basic data structures and their encoding in the polyadic $\pi$-calculus. These data structures are important for the translation of the language School $\pi$, The encoding is partly standard, but some parts e.g. the persistent numbers and the queue, are our own.

3.1 The monadic $\pi$-calculus

The monadic $\pi$-calculus is a descendant of the process algebra CCS. In the process algebra CCS the mobility could not be expressed directly. By the mobility we mean the ability to change the structure of a process and the linkage between
processes. In π-calculus this mobility is achieved by using channel names as data that can be sent along a channel.

3.1.1 Syntax

The basic assumption in π-calculus is the existence of an infinite set $\mathcal{N}$ of channel names. We let lower-case letters range over names e.g. x, y, z are channel names. We let upper-case letters range over process expressions. The process expressions are defined as follows:

\[
P ::= N \quad \text{normal process}
\]

\[
P \parallel P \quad \text{parallel composition}
\]

\[
(\nu x)P \quad \text{restriction}
\]

\[
A(x_1, \ldots, x_n) \quad \text{defined agent}
\]

\[
N ::= 0 \quad \text{null agent}
\]

\[
x!y . P \quad \text{output prefix}
\]

\[
x?y . P \quad \text{input prefix}
\]

\[
N + N \quad \text{summation}
\]

A normal process or so-called guarded summation consists of either null agents – process without reduction, or output prefix, or input prefix – it binds the input variable $y$, or the summation operator. A process is a normal process, two processes in parallel composition, a restriction or a defined agent. Two processes in parallel can act independently or engage in communication. Restriction binds the restricted variable $x$ in the process $P$. Defined agent substitutes the channel names in the agent definition. In agent definitions, all the variables must be distinct!

The precedence of the operators is:

\[
\begin{align*}
\text{restriction} \quad > \\
\text{input prefix} \quad > \\
\text{output prefix} \quad > \\
\text{summation} \quad > \\
\text{parallel composition}
\end{align*}
\]

We also define free names of process $\mathcal{P} – \text{fn}(\mathcal{P})$ – to be the channel names in $\mathcal{P}$ not bounded by a restriction or by input prefix. The bound names of process $\mathcal{P}$ will be written as $\text{bn}(\mathcal{P})$. Just as in the λ-calculus, we shall not distinguish between agents that are $\alpha$-convertible i.e. between those that differ only by a
change of bound names. We shall write $P = Q$ if $P$ and $Q$ are $\alpha$-convertible. We write $P[x_1/y_1 \ldots x_n/y_n]$ for the simultaneous substitution of $y_i$ for all free occurrences of $x_i$ (for $i=1..n$) in the process $P$ with change of bound names if necessary to prevent any of the new names $x_i$ from becoming bound in $P$.

### 3.1.2 Examples

In our early examples we show the basic ideas in graphs. In the first graph we have three agents $P$, $Q$, and $R$.

The agent $P$ has a link $x$ to $R$, and wishes to pass $x$ along its link $y$ to $Q$. Agent $Q$ is willing to receive it. Thus $P$ may be $y!x.P'$ and $Q$ may be $y?z.Q'$. In this case, the transition is

$$y!x.P' \mid y?z.Q' \mid R \rightarrow P' \mid Q'\{x/z\} \mid R$$

If the channel names were private to any one of the agents $P$, $Q$, $R$ the situation would be almost the same. The above system behaves differently if we assume that $Q$ owns a private link $x$ to some agent. In this case, when the agent $P$ sends the channel $x$ to the agent $Q$ it intrudes the scope of the private channel $x$ – this phenomenon is called scope intrusion. To avoid confusion of channels the private channel must be renamed. In the graph below, the scope of private names is shown by the dotted line and the private names are printed inside the dotted line.

The above graph can be written as a transition between process expressions:
Scope intrusion is analogous to the avoidance of the capture of bound variables in the \(\lambda\)-calculus (\(\alpha\)-conversion). The transition rules must ensure that no confusion of names is possible. Just as in the \(\lambda\)-calculus, \(\alpha\)-conversion (i.e. change of bound variables) is enforced in such cases – in \(\pi\)-calculus, it is the change of private names.

In the previous examples, the agent \(P\) has always sent the agent \(Q\) a public name. The next example shows what happens when the agent \(P\) sends a private name. The example also includes the scope intrusion – the agent \(Q\) owns a public name \(x\), which intrudes the scope of the new received private name \(x\). The private name must be renamed and then it can be sent. When the agent \(Q\) receives the private name its scope is extended to include the agent \(Q\) – this is called scope extrusion.

The dotted lines show the scope of the private names printed inside the dotted line. The graph is easily rewritten into transition of process expressions:

\[
(y!x.P' \mid (vx)(y?z.Q' \mid \ldots) \mid R) \\
\rightarrow P' \mid (vx')(Q'\{x'/x\}\{z'/z\} \mid \ldots) \mid R
\]

3.1.3 Semantics

The semantics is defined in terms of a labelled transition system. To make the reduction rules simpler we define structural congruence \(\equiv\). Structural congruence is the smallest congruence relation over \(P\) such that the following laws hold:

1. Agents (processes) are identified if they only differ by a change of bound names
2. \((N/\equiv; +; 0)\) is a symmetric monoid, i.e. the following rules hold

\[
P + Q \equiv Q + P \quad (P + Q) + R \equiv P + (Q + R) \quad P + 0 \equiv P
\]
3. \((P; |; 0)\) is a symmetric monoid, i.e. the following rules hold

\[ P \mid Q \equiv Q \mid P ; (P \mid Q) \mid R \equiv P \mid (Q \mid R) ; P \mid 0 \equiv P \]

4. \((v x) 0 \equiv 0 ; (v x) (v y) P \equiv (v y) (v x) P\)

5. If \(x \not\in \text{fn}(P)\) then \((x) (P \mid Q) \equiv P \mid (x) Q\)

We introduce a notation to shorten the code – the 4\(^{th}\) rule allows us to abbreviate two or more restrictions that follow each other e.g. instead of \((v x) (v y) (v z)\) we write \((v x, y, z)\).

The second step in defining the semantics of the \(\pi\)-calculus is the definition of the reduction relation \(\rightarrow\). The only axiom for the reduction relation \(\rightarrow\) is called communication and it is defined over normal processes:

\[
\text{COMM: } (\ldots + x!y.P) \mid (\ldots + x?z.Q) \rightarrow P \mid Q^{y/z}
\]

To finish the definition of the semantics we define three inference rules. The first rule says that reduction can happen independently underneath parallel composition.

\[
\text{PAR: } P \rightarrow P' \\
\binom{P \mid Q \rightarrow P' \mid Q}{P \rightarrow P'}
\]

The second rule says that when reduction can happen, then the same reduction can be achieved underneath restriction.

\[
\text{RES: } P \rightarrow P' \\
\binom{(v x)P \rightarrow (v x)P'}{(v x)P \rightarrow (v x)P'}
\]

The last rule says that process expressions identified by the structural congruence have the same reduction.

\[
\text{STRUCT: } Q \equiv P \rightarrow P' \rightarrow P' \equiv Q' \\
\binom{Q \rightarrow Q'}{Q \equiv P \rightarrow P' \rightarrow P' \equiv Q'}
\]

We show one example of reduction in detail, while we demonstrate so-called molecular actions. Suppose we have three agents. The agent \(P\) wants to send a pair of values \((u \; v)\) to some other agent. The agents \(Q\) and \(R\) are waiting to receive this pair.
CHAPTER 3. THE $\pi$-CALCULUS

We put these agents into parallel composition ($P \mid Q \mid R$) to make the communication possible. The unfortunate thing is that the agents $Q$ and $R$ can interfere with each other’s communication to the agent $P$. This can lead to unwanted state when the agent $Q$ receives one of the values and the agent $R$ receives the other one. For example:

$$P \equiv x!u.x!v.P'$$
$$Q \equiv x?p_1.x?p_2.Q'$$
$$R \equiv x?p_1.x?p_2.R'$$

We need the protection of the transmission of a pair of values against breaking apart. This protection can be achieved by using a private name. The agent $P$ will send the private name to the receiver of the pair of values and then delegate a process to send the pair along this private channel. Now, we have the following processes:

$$P = (vw)(x!w.P' \mid w!u.w!v.0)$$
$$Q = x?z.z?p_1.z?p_2.Q'$$
$$R = x?z.z?p_1.z?p_2.R'$$

The system consisting of three agents $P$, $Q$, $R$ can make two different sets of transitions each one arriving in the proper state. We show the complete set of transitions in which the agent $Q$ receives the pair of values. The second possible set of transitions in which the agent $R$ receives the pair is analogous.

$$P \mid Q \mid R = Q \mid P \mid R$$

We put these agents into parallel composition ($P \mid Q \mid R$) to make the communication possible. The unfortunate thing is that the agents $Q$ and $R$ can interfere with each other’s communication to the agent $P$. This can lead to unwanted state when the agent $Q$ receives one of the values and the agent $R$ receives the other one. For example:

$$P \equiv x!u.x!v.P'$$
$$Q \equiv x?p_1.x?p_2.Q'$$
$$R \equiv x?p_1.x?p_2.R'$$

We need the protection of the transmission of a pair of values against breaking apart. This protection can be achieved by using a private name. The agent $P$ will send the private name to the receiver of the pair of values and then delegate a process to send the pair along this private channel. Now, we have the following processes:

$$P = (vw)(x!w.P' \mid w!u.w!v.0)$$
$$Q = x?z.z?p_1.z?p_2.Q'$$
$$R = x?z.z?p_1.z?p_2.R'$$

The system consisting of three agents $P$, $Q$, $R$ can make two different sets of transitions each one arriving in the proper state. We show the complete set of transitions in which the agent $Q$ receives the pair of values. The second possible set of transitions in which the agent $R$ receives the pair is analogous.

$$P \mid Q \mid R = Q \mid P \mid R$$
The atomic action $x!w$ together with process $w!u.w!v.0$ – which becomes accessible by the atomic action – is usually called a *molecular action*. Molecular actions are a very important tool for encoding data structures as processes.

### 3.1.4 Data structures

To show some standard data structure we decided to present the encoding of lists and some of the basic operations with lists. A list $L$ is represented as an agent $[L](r)$ with parameter $r$ being the channel where the agent can send the list $L$ piecemeal. We define the list in terms of constant names $\text{nil}$ and $\text{cons}$ that represent the list constructors. The encoding is:

$$[\text{nil}](r) := r!\text{nil}$$

$$[\text{cons } L_1 L_2](r) := (v \ x, y)(r!\text{cons}.r!x.r!y \mid [L_1](x) \mid [L_2](y))$$

The molecular actions used in this encoding are the atomic action $r!x$ together with the process $[L1](x)$ and the action $r!y$ together with the process $[L2](y)$. To show some basic list operations we present a process that can copy a list from one channel to another. The agent $\text{copy}$ selects its actions upon the structure of the list. An empty list causes no trouble. A cons cell is taken apart and the new cons cell is created from the same element and the copy of the rest of the list.

$$\text{copy}(1, r) := 1?\text{nil}.[\text{nil}](r)$$

$$+ 1?\text{cons}.1?x.1?y.$$

$$(\text{vc})(r!\text{cons}.r!x.r!c \mid \text{copy}(y,c))$$

When we have defined copying, we can easily define concatenation of two lists. Concatenation acts as an extended copy agent that copies the first list and then after the first list was emptied it calls the copy agent to copy the second list.

$$\text{append}(l_1, l_2, r) := 1?\text{nil}.\text{copy}(l_2, r)$$

$$+ 1?\text{cons}.1?x.1?y.$$  

$$(\text{va})(r!\text{cons}.r!x.r!a \mid \text{append}(y, l_1, a))$$

### 3.2 The polyadic $\pi$-calculus

In the polyadic $\pi$-calculus, more that one channel name can be sent along a channel. This may look just like an abbreviation for the molecular actions, but there are several reasons for the polyadicity. One of them is the use of abstractions and the other one is the ability to assign a type to the polyadic $\pi$-calculus.
expression. In this section, we show only the polyadic $\pi$-calculus and the type system will be shown in Chapter 4.

3.2.1 Syntax

The syntax of the polyadic $\pi$-calculus is enriched significantly. It introduces two new syntactic categories, i.e. abstractions and concretions.

Processes: $P ::= N \mid P \mid P \mid (\nu x)P \mid *P$

Normal processes: $N ::= x?F \mid x!C \mid 0 \mid N + N$

Abstractions: $F ::= P \mid (\lambda x)P \mid (\nu x)P$

Concretions: $C ::= P \mid [x]P \mid (\nu x)P$

Process expression or process is a normal process, two processes in the parallel composition, channel name restriction or a new construct – replication. A replicated process is one that can spawn infinitely many copies of itself.

Normal processes are almost the same as in the monadic calculus – null process, sum of processes and new prefix forms. The new prefix $x?F$ waits on the channel $x$ for the instantiation of the abstraction $F$. Prefix $x!C$ “sends” a concretion $C$ along the channel $x$. The word sends is put in quotation marks because it does not mean that the whole concretion is sent. Only the channel names that form the first part of the concretion are sent.

Abstractions are useful in agent definitions. As we define an agent in the monadic $\pi$-calculus – we use a definition to make the agent parametric. In the polyadic $\pi$-calculus, we use abstractions to parameterise the agent. In stead of

$$A(x_1, \ldots, x_n) ::= P$$

in the polyadic $\pi$-calculus we write

$$A ::= (\lambda x_1) \ldots (\lambda x_n)P$$

This notation is reminiscent of a function definition in the $\lambda$-calculus. The important difference is that while in the $\lambda$-calculus the parameters can be instantiated to any value, even a complex one, in the $\pi$-calculus the parameters are instantiated only to channel names!
3.2.2 Semantics

The first five rules for structural congruence are the same as in the monadic case. We add four new rules. The rule number six is for replication to allow the replicated process to create copies of itself.

6. \( *P \equiv P \mid *P \)

The next three rules deal with abstractions and concretions.

7. \((vy)(\lambda x)F = (\lambda x)(vy)F\) \(\quad(x \neq y)\)

8. \((vy)[x]C = [x](vy)C\) \(\quad(x \neq y)\)


These rules allow the abstractions and concretions to be converted to a standard form. Taking an abstraction \(F\) and using rules 1 and 7 we can push all the restrictions inwards and we get a standard form for abstractions:

\[ F = (\lambda x_1) \ldots (\lambda x_n)P \]

Using rules 1 and 8 with a concretion \(C\) we can pull all the channel name restrictions outwards and all the other restrictions inwards and we get a standard form for concretions:

\[ C = (v \ x_{i_1}, \ldots, x_{i_\ell})[x_1] \ldots [x_n]P \quad i_1 \in \{1 \ldots n\} \]

We also introduce a new notation for normal processes that have a prefix form and include abstraction resp concretion in the standard form. We often write

\[ x?[x_1 \ldots x_n].P \text{ instead of } x?(\lambda x_1)\ldots(\lambda x_n)P \text{ and} \]

\[ (v \ x_{i_1}, \ldots, x_{i_\ell})x![x_1 \ldots x_n].P \text{ instead of } \]

\[ x!(v \ x_{i_1}, \ldots, x_{i_\ell})[x_1 \ldots x_n].P \]

In addition, when the abstraction resp concretion is just a process we often write

\[ x![] .P \text{ instead of } x!P \text{ and} \]

\[ x?[] .P \text{ instead of } x?P \]

to emphasise that the abstraction resp concretion is empty, i.e. it receives res. sends an empty tuple.

We must modify the reduction relation rule \(\text{COMM}\) for the polyadic communication. To follow the definition of the new rule \(\text{COMM}\) in [Mil91] we first define \(\text{pseudo-application } F \cdot C\). When an abstraction \(F\) and a concretion \(C\), both in stan-
standard forms, have equal arity i.e. \( F = (\lambda x_1) \ldots (\lambda x_n) P \) and \( C = (v \ y_1, \ldots, y_{i_b}) [y_1] \ldots [y_n] Q \) then
\[
F \cdot C = (v \ y_1, \ldots, y_{i_b}) (P[y_1/x_1] \ldots y_n/x_n) \mid Q
\]

With the help of pseudo-application, we can easily modify the \textsc{comm} rule.

\textsc{comm}: \( (\_ + x . C) \mid (\_ + x . F) \rightarrow F \cdot C \)

This rule together with the inference rules \textsc{par}, \textsc{res} and \textsc{struct} from the monadic calculus semantics operate over processes. The reduction relation \( \rightarrow \) over processes is now the least relation that satisfies these rules.

A new important feature appears in the polyadic \( \pi \)-calculus. In case the abstraction and concretion located at the same location, i.e. willing to communicate, do not have the same arity such ill formed process fails!

### 3.3 Data structures

Standard data structures can be encoded in the polyadic \( \pi \)-calculus in many different ways. We show in this section encoding of data structures that is persistent in the sense that it can be accessed infinite number of times.

#### 3.3.1 Functions as processes

It is very common to use processes in a ‘functional way’, i.e. process accepts a number of arguments and after some computation returns a result. The only way to return the result is to send it along a channel. This means that process-function must receive one additional argument, i.e. the result channel. Then a function can be represented as process:

\[
\text{func}[x \ r] \ldots r![	ext{result}]
\]

If the function is to be called more then once the above process must be replicated. The replicated form is our standard process-function form.

\[
*\text{func}[x \ r] \ldots r![	ext{result}]
\]

#### 3.3.2 Boolean

In the \( \lambda \)-calculus, the truth-values are represented by the terms \( \lambda t \lambda f t \) and \( \lambda t \lambda f f \).

With the help of abstractions, we can transform this directly into the \( \pi \)-calculus.
This is so-called unlocated representation of data. To use these data values we must locate them in some location. The located value is for example \( \pi \text{true} \). To make the value persistent the process representing the located value must be replicated. In addition, we create two process-functions that return a persistent truth-value. Process-functions \( \pi \text{true} \) and \( \pi \text{false} \) create a location of the value and then return it together with creating an appropriate persistent located value.

\[
\begin{align*}
\ast \text{True}&[r].(vb)(r![b] | \ast \text{b} \text{true}) \\
&= \ast \text{True}&[r].(vb)(r![b] | \ast \text{b} [t \text f].t![])
\end{align*}
\]

\[
\begin{align*}
\ast \text{False}&[r].(vb)(r![b] | \ast \text{b} \text{false}) \\
&= \ast \text{False}&[r].(vb)(r![b] | \ast \text{b} [t \text f].f![])
\end{align*}
\]

Processes that are shown as congruent are just written in the abbreviated syntax for abstractions and concretions in the standard form.

### 3.3.3 Conditional

We can create a concretion that can perform a case-analysis of truth-values. The test channels \( t \) and \( f \) are private to make the case-analysis closed and insensible to other communications.

\[
\begin{align*}
\pi \text{case}(P, Q) &:= (v \ t, \ f)[t \ f](t?[].P + f?[].Q) \\
\text{t, f} &\in \text{fn}(P, Q)
\end{align*}
\]

Using the pseudo-application, we can see how the case analysis works.

\[
\begin{align*}
\pi \text{true}\pi \text{case}(P, Q) &= (v \ t, \ f)(t![]) \ | \ (t?[].P + f?[].Q)) \\
&\rightarrow (v \ t, \ f)P = P
\end{align*}
\]

Now the classical conditional form can be easily expressed as a so-called co-located case-analysis. The name ‘co-located’ comes from the fact that in the original syntax, there are names and co-names of channels – names represent input and co-names represent output.

\[
\begin{align*}
\pi \text{if b then P else Q} &= \pi b!\pi \text{case}(P, Q) \\
&= (v \ t, f)(b![t \ f].(t?[].P + f?[].Q))
\end{align*}
\]
3.3.4 Numerals

Numerals in the π-calculus are very similar to the Church Numerals used in the λ-calculus. The difference is only in the parameters that are not λ-terms but channel names and therefore the parameters must be treated appropriately. The unlocated numerals are:

\[
\text{zero} := (\lambda s)(\lambda z)z!
\]
\[
n := (\lambda s)(\lambda z)(s!)^n z!
\]

The process-functions that return the appropriate persistent numerals are:

\[
*\text{Zero?}[r].(v1)(r! [1] \mid *1?\text{zero})
\]
\[
*\text{N?}[r].(v1)(r! [1] \mid *1?n)
\]

The basic operations succ resp pred that return a successor resp predecessor of a numeral can be encoded as:

\[
*\text{succ?}[x\ r].(v1)(r! [n] \mid *n[s\ z].s![]\cdot x! [s\ z])
\]
\[
*\text{pred?}[x\ r].(v1)(r! [n] \mid *n[s\ z].(vq)(x! [q\ z].q?[].*q?[].s![]))
\]

The successor only adds one more s![] prefix. The predecessor is slightly more complex because it instantiates the received numeral. Then it consumes the first s![] prefix and finally returns all the remaining s![] prefixes; when zero is reached it is returned by the original numeral – this guarantees that the predecessor of zero is zero.

Addition of two numerals is simple. It allows the first numeral to be transmitted and when its z![] prefix is reached it allows the second numeral to continue.

\[
*\text{add}[x\ y\ r].(v1)(r! [n] \mid *n[s\ z].(vc)(x! [s\ c].c?[].y! [s\ z]))
\]

Multiplication creates a numeral that instantiates the first parameter and then it transmits the second parameter n-times where n is the value of the first parameter. Zero prefix z![] of the second parameter is never send out but is used to start the new iteration instead.
3.3.5 Reference cell

As Pierce in [Pie95] pointed out: “The pi-calculus is a quintessentially imperative (i.e. non-functional) language, in the sense that we do not expect to get the same result each time if we query a process repeatedly over a channel. A typical example is a ‘reference cell object,’ which maintains a single piece of state, updating it in response to messages sent over a channel w and reporting its current value in response to messages sent over a channel r.”

We use a channel \( V \) to update the value and a channel \( W \) the read the current value instead of channels w and r. Reference cell is a simple process that uses a private channel \( Q \) to store the current value. It returns a tuple consisting of two channels representing the available operations. An operation – represented by a process – reads the current state and updates it (set) or leaves it unchanged (get) appropriately.

\[
*\text{ref?}[\text{init } r].
\]

\[
(\nu \text{ content}, \text{set}, \text{get})
\]

\[
(*\text{(set?}[x \ r]. \text{content?}[\text{old}]. \text{(content!}[x] \mid \text{r}[\text{!}]))
\]

\[
+\text{get?[r]. content?[x]. \text{(content!}[x] \mid \text{r}[\text{x}])}
\]

\[
\mid \text{content!}[\text{init}]
\]

\[
\mid \text{r!}[\text{set get}]
\]

When an operation has read a value from the private channel content, it temporarily blocks all the other operations. This means that at most one operation can be active. Hence, this mechanism avoids any concurrent interference.

3.3.6 Queue

A queue is a standard FIFO structure. The queue has two basic operations put and get. Put operation accepts one parameter and put the channel to the end of the current queue. Get operation accepts a result channel and sends the first element along it.
*queue?[r].

(v last, current, put, get)
(*put?[c].
   (vack)(last![ack].
     (last?[a].ack?[] . current?[x].a![[]
     +ack?[] . (last?[a]. current?[x].a![[]
     +current?[x].last?[ack].ack![])))
   + get?[r].current?[c].r!c)
   | last?[ack].ack![] | r![put get])

Internally the queue is a kind of a linked list. The elements in the list are connected via acknowledgement channel ack – each element has its own private channel ack. The last element in the list is always ready to receive the next element along the channel last. The channel last serves internally for adding an element to the end of the queue. The channel current holds the information stored in each element. The information in an element becomes accessible only after an acknowledgement comes along the private channel ack. The process last?[ack].ack![] serves as an empty. Unfortunately, this simple queue does not have the required property that after two elements are added and only one is extracted the extracted one is the same as the first inserted element.

(v x, y, r)

(queue?[r].r?[put get].put?[x].put?[y].get?[r].r?[e])

It means that e can be equal to x but it can also be equal to y. This problem is caused by the definition of operation put. Operation put is not forced to insert an element to the internal list immediately. It can wait for any time. It is even possible that an element is never inserted. To give as simple solution as possible we allow only one element to be inserted at a time.

*queue?[r].

(v available, last, current, put, get)
(*available?[[].
   (put?[c].
      (vack)(last![ack].available![].
        (last?[a].ack?[] . current?[x].a![]
        +ack?[] . (last?[a]. current?[x].a![]
        +current?[x].last?[ack].ack![])))
        +get?[r].current?[c].available![] . r!c)
        | last?[ack].ack![] | available![] | r! [put get])
The channels put and get are protected by the channel available. When operator put is called, it will make the queue available to any subsequent modification only after the new element is inserted at the end of the internal list. The operator get makes the queue available just after it has read the element at the head of the queue.
Chapter 4

The PICT language

This chapter is devoted to the definition of the language PICT. Before we present the PICT language itself we turn our focus to the basic features of the asynchronous $\pi$-calculus. In section 4.1, we show the basic differences between the synchronous and the asynchronous $\pi$-calculus.

Then section 4.2 introduces a type system developed by Turner in [Tur95]. This type system is a base for the type system used in language PICT.

The language PICT is based on the typed asynchronous $\pi$-calculus. Section 4.3 closes this chapter with a description of features of the language PICT, which are important in this thesis. Section 4.3 is based on [PT95] and [PT97].

4.1 The asynchronous $\pi$-calculus

The monadic and the polyadic $\pi$-calculus described in Chapter 3 use a synchronous handshake protocol during communication. In 1991, Honda and Tokoro investigated in their paper [HT91] an asynchronous object calculus that is a sub-calculus of the monadic $\pi$-calculus. This paper initiated studying asynchronous mobile calculi. The calculus presented in [HT91] suffers from its monadicity and the lack of summation. Processes represented in this calculus tend to be complex. Even a simple sending of several channel names from one agent to the other agent is intricate and it requires about twice the number of reduction steps then in the synchronous $\pi$-calculus.

The major difference between the synchronous $\pi$-calculus and the asynchronous $\pi$-calculus is the output. While in the synchronous calculus the output prefix is a
standard prefix in the asynchronous calculus, the asynchronous output is a special prefix that restricts the process that follows it to the null agent. The other often-employed difference is that the asynchronous calculus is used without the standard summation.

4.2 Typing

Turner presented in [Tur95] a polymorphic type system for the $\pi$-calculus, which is an evolution of the sort system presented in [Mil91]. We show in this section the basic rules for the polymorphic channel types. We left out the rules for recursive types, because we do not use recursive types in this diploma thesis. The proposed type system also includes a type inference algorithm, which can automatically infer types of $\pi$-calculus expressions.

4.2.1 Basics

The only data in the $\pi$-calculus are channels. Therefore, we can assign types only to channels. The type variables are included to allow type inference, recursion and polymorphism.

$$\delta ::= \alpha$$

(type variable)

$$\mu\alpha.\delta$$

(recursive type)

$$\uparrow[\alpha_1 \ldots \alpha_m ; \delta_1 \ldots \delta_n]$$

(polymorphic channel type)

We also need two definitions to make the rules clear. A type context $\Delta$ is a sequence of bindings.

$$\Delta ::= x_1:\delta_1, \ldots, x_n:\delta_n$$

The expression $\Delta(x)$ denotes the type associated with $x$ in $\Delta$, and its definition is:

$$(\Delta, x:\delta)(x) ::= \delta$$

$$(\Delta, y:\delta)(x) ::= \Delta(x)$$

$$(\Delta)(x) ::= \text{undefined}$$

We also define function ftv($\Delta$), which returns a set of free type variables included in those types contained in $\Delta$. 

4.2.2 Rules

The typing rules are in form $\Delta \vdash P$ where $\Delta$ is a type context that holds the types of the free variables of the process $P$. The rule $\Delta \vdash P$ is read as: the process $P$ uses its free variables consistently with the types given in $\Delta$.

The simplest typing rule is for the simplest $\pi$-calculus expression. The null agent is consistent with every type context $\Delta$.

$$\text{NIL}: \quad \Delta \vdash 0$$

Processes in parallel composition must use their variables in the same consistent way. When both processes use the same channel, the following rule ensures that the channel has the same type.

$$\text{PRL}: \quad \Delta \vdash P \quad \Delta \vdash P \quad \Delta \vdash P \upharpoonright Q$$

Very similar rule is used for summation. The rule guarantees that every branch of summation is consistent with the same type context – this is important, because we do not know which one will be executed.

$$\text{SMT}: \quad \Delta \vdash P \quad \Delta \vdash P \quad \Delta \vdash P + Q$$

The replication operator makes an arbitrary number of copies of the same process and because the type system does not check the multiplicity of channel use, the type of replicated process depends only on the original process.

$$\text{REPL}: \quad \Delta \vdash P \quad \Delta \vdash P^{*}$$

The rule for restriction forces the type assigned to the restricted variable to be a channel type.

$$\text{RES}: \quad \Delta, x: \uparrow[\alpha_1 \ldots \alpha_m \uplus \delta_1 \ldots \delta_n] \vdash P \quad \Delta \vdash (vx: \uparrow[\alpha_1 \ldots \alpha_m \uplus \delta_1 \ldots \delta_n])P$$

Polymorphic input rule only checks that the type of channel $c$ has a sufficient type structure for the body of input. If $m=0$ this rule yields the monomorphic type rule.
In the case of polymorphic output, the rule checks that the values sent along the channel \( c \) are substitution instances of the types specified in the type of the channel \( c \).

\[
\Delta(c) = \uparrow[\alpha_1 \ldots \alpha_m ; \delta_1 \ldots \delta_n] \quad \Delta, x_1 : \delta_1, \ldots, x_n : \delta_n \vdash P
\]

\[
\Delta \vdash c?[^{\downarrow}\alpha_1 \ldots \alpha_m ; x_1 \ldots x_n].P
\]

The type system presented in [Tur95] is sound, i.e. well-typed processes never fail – if \( \Delta \vdash P \) then \( P \) does not fail.

### 4.3 The **P**ICT language

The language **P**ICT is based on the polymorphic asynchronous \( \pi \)-calculus. The definition of the language does not include summation. The choice (summation) is implemented as a library module. The library also includes a special choice operator called a replicated choice – the replicated choice represents a replicated normal process constructed only with summation e.g. \( \ast(P + Q) \). The language also restricts the use of replication only to input expressions – replicated input.

#### 4.3.1 Core language syntax

The language **P**ICT provides a special syntax for built-in types: strings, characters, integers and booleans. These built-in types are implemented as abstract data types to give the compiler the maximum freedom of implementation. The built-in abstract data types include standard operations represented as functions-processes, e.g. \( + \) for integers, addition is represented as \( +[10 \ldots r] \).

The basic syntactic category is a value. The value can be sent along a channel. The meaning of each construct is included at the end of each line.
Val ::= id
[ Label Val ... Label Val ]
{ Type } Val
( rec : T Val )
String
Char
Int
Bool

Labels used in records are defined as:

Label ::= (empty)

id =

On the receivers side the value can be decomposed. A pattern specifies the decomposition and binding of variables and type variables.

Pat ::= id : Type
    _ : Type
    id : Type @ Pat
    [ Label Pat ... Label Pat ]
    { id < Type } Pat
    ( rec : T Pat )

Abstractions in the PICT are defined using patterns.

Abs ::= Pat = Proc

Processes are defined in a standard manner. The only interesting point is that arbitrary values are allowed on the left of !, ?, and ?. It is the responsibility of the type system to allow only channel names to be used in the subject position.

Proc ::= Val ! Val
    Val ? Abs
    Val ?* Abs
    ( Proc | Proc )
    ( Dec Proc )
    if Val then Proc else Proc
Declaration of a new channel must be annotated with an explicit type. The local declaration is used as a restriction.

\[
\text{\textit{Dec}} \quad ::= \quad \text{\textit{new id : Type}}
\]

Types are syntactically defined:

\[
\text{\textit{Type}} \quad ::= \quad \land \text{\textit{Type}} \quad \quad \quad \text{input/output channel}
\]
\[
\land \text{\textit{Type}} \quad \quad \quad \text{output-only channel}
\]
\[
? \text{\textit{Type}} \quad \quad \quad \text{input-only channel}
\]
\[
\{ \text{\textit{id < Type}} \} \text{\textit{Type}} \quad \quad \quad \text{package type}
\]
\[
[ \text{\textit{Label Type}} \ldots \text{\textit{Label Type}} ] \quad \quad \quad \text{record type}
\]
\[
\text{\textit{id}} \quad \quad \quad \quad \quad \quad \quad \quad \text{type identifier}
\]
\[
\setminus \text{\textit{id : Kind = Type}} \quad \quad \quad \text{type operator}
\]
\[
( \text{\textit{Type Type}} ) \quad \quad \quad \text{type abstraction}
\]
\[
( \text{\textit{rec id : Kind = Type}} ) \quad \quad \quad \text{recursive type}
\]
\[
\text{\textit{Top : Kind}} \quad \quad \quad \quad \quad \quad \quad \quad \text{maximal type}
\]
\[
\text{\textit{String}} \quad \quad \quad \quad \quad \quad \quad \quad \text{string type}
\]
\[
\text{\textit{Char}} \quad \quad \quad \quad \quad \quad \quad \quad \text{character type}
\]
\[
\text{\textit{Int}} \quad \quad \quad \quad \quad \quad \quad \quad \text{integer type}
\]
\[
\text{\textit{Bool}} \quad \quad \quad \quad \quad \quad \quad \quad \text{boolean type}
\]

Each type has its own kind. Kinds distinguish type operators and types.

\[
\text{\textit{Kind}} \quad ::= \quad ( \text{\textit{Kind \rightarrow Kind}} ) \quad \quad \quad \text{kind of type operator}
\]
\[
\text{\textit{Type}} \quad \quad \quad \quad \quad \quad \quad \quad \text{kind of types}
\]

### 4.3.2 Semantics

As usual, we define the structural congruence, first. The structural congruence copies the definition of the structural congruence in the $\pi$-calculus (see subsection 3.1.3). The rules are different in the used syntax, only.

\[
\text{\textit{STR-COMM}}: \quad (e_1 \mid e_2) = (e_2 \mid e_1)
\]
\[
\text{\textit{STR-ASSOC}}: \quad ((e_1 \mid e_2) \mid e_3) = (e_1 \mid (e_2 \mid e_3))
\]
\[
\text{\textit{STR-EXTRUDE}}: \quad x \notin \text{fn}(e_2) \quad \implies \quad ((\text{new } x:T \ e_1) \mid e_2) = (\text{new } x:T \ (e_1 \mid e_2))
\]
Standard substitution in the language PICT is written as \( \{ x \mapsto y \}^p \) instead of \( p[^{x/y}] \) used in the \( \pi \)-calculus. The important feature of the language PICT is pattern matching. When a pattern is successfully matched against a value, the result is a substitution defined as follows:

\[
\{ x : T \mapsto v \} = \{ x \mapsto v \}
\]

\[
\{ _: T \mapsto v \} = \{ \}
\]

\[
\{(x:T@p) \mapsto v\} = \{x \mapsto v\} \cup \{p \mapsto v\}
\]

\[
\{(\text{rec}:T \ p) \mapsto (\text{rec}:S \ v)\} = \{p \mapsto v\}
\]

\[
\{(\text{X}<S \ p) \mapsto \{T\}v\} = \{(x \mapsto T) \cup \{p \mapsto v\}
\]

\[
\{[1_1\llbracket p_1 \ldots \llbracket p_n \rrbracket \mapsto \llbracket 1_1 \llbracket v_1 \ldots \llbracket v_n \ldots \}\} = \{p_1 \mapsto v_1\} \cup \ldots \cup \{p_n \mapsto v_n\}
\]

The reduction relation is defined in terms of the substitution and matching. The important rule for communication specifies that communication happens only in the case the input pattern matches the output value.

\[
\text{RED-COMM: } \{ p \mapsto v \} \text{ defined}
\]

\[
(x!v \mid x?p = e) \rightarrow \{ p \mapsto v \} e
\]

In the case of replicated input the rule is almost the same, except it keeps the original replicated process.

\[
\text{RED-RCOMM: } \{ p \mapsto v \} \text{ defined}
\]

\[
(x!v \mid x?^\ast p = e) \rightarrow (\{ p \mapsto v \} e \mid x?^\ast p = e)
\]

The next two rules are the same as the \( \pi \)-calculus reduction rules RES and PAR. The only change is the change of syntax.

\[
\text{RED-DEC: } e_1 \rightarrow e_2
\]

\[
(d \ e_1) \rightarrow (d \ e_2)
\]

\[
\text{RED-PRL: } e_1 \rightarrow e_3
\]

\[
(e_1 \mid e_2) \rightarrow (e_3 \mid e_2)
\]

The reduction relation is extended with rules for conditional construct. These rules are straightforwardly defined as:

\[
\text{RED-If-T: } \text{If true the } e_1 \text{ else } e_2 \rightarrow e_1
\]

\[
\text{RED-If-F: } \text{If false the } e_1 \text{ else } e_2 \rightarrow e_2
\]

The language PICT includes powerful type system based on the type system presented in section 4.2. The PICT type system is extended with sub-typing. The whole type system is quite complex and goes beyond the scope of this diploma.
thesis. The understanding of the rules presented in section 4.2 and general knowledge of sub-typing used in functional languages is sufficient for the purposes of this thesis. For an interested reader, the complete definition of the language PICT is available electronically as a part of the PICT distribution.

4.3.3 Derived forms

The language PICT also includes several high-level derived forms. These forms are encoded in the core language. In this subsection, we present only few of the derived forms – those used in this thesis.

The simplest derived form is a more compact form for a sequence of declarations.

\[(d_1 \ldots d_n \text{ e}) \Rightarrow (d_1 \ldots (d_n \text{ e})\ldots) \quad \text{(TR-DECSEQ)}\]

The language PICT allows process abstractions to be defined. These abstractions can be recursive and mutually recursive. A group of mutually recursive definitions is translated by the rule TR-DEF.

\[(\text{def } x_1 a_1 \ldots \text{ and } x_n a_n \text{ e}) \Rightarrow (\text{new } x_1 \ldots \text{ new } x_n (x_1 ? a_1 | \ldots | x_n ? a_n | \text{ e})) \quad \text{(TR-DEF)}\]

As an example of the group of two mutually recursive process, we show two abstractions f and g.

\[\text{def } f \ [x \ r] = \ldots \ g! [a \ b] \ldots \text{ and } g \ [y \ s] = \ldots \ f! [a \ b] \ldots \ f![1 \ 2] \]

Since anonymous abstractions are useful for higher-order programming, see example in the next subsection, the language defines a special form that looks similar to lambda abstractions in the \(\lambda\)-calculus.

\[\lambda a \Rightarrow (\text{def } x \ a \ x) \quad \text{(TR-ANONABS)}\]

4.3.4 Example

We present one example that is necessary in the next chapters. We define a reference cell object. For a comparison with the encoding of the reference cell already shown in the \(\pi\)-calculus, see subsection 3.3.5.

\[\text{type } \text{Ref} = \\lambda T: \text{Type} = \{\text{set: } ?[T ![]] \text{ get: } ?![T]]\]
This example shows the syntax of replicated choice. It is constructed from a channel name (content) and a sum ($) of processes. These processes are built using operator and consist of a channel name and an anonymous abstraction that includes the code executed when the communication through the appropriate channel occurs. The whole replicated choice construction is then send along the channel start, which is activated by the first reception along the control channel (content).
Chapter 5

Language School$_{98}$

This chapter presents a top-down approach in building a new concurrent object-oriented language. We design a language named School$_{98}$ and subsequently present its semantics in terms of the polyadic $\pi$-calculus. The goal of this chapter is to design a flexible simple language with minimum language constructs that can be easily enriched with other features. The translation must be such that an addition of new features will be as painless as possible.

We discuss firstly the syntactic constructs that we use in our language together with their appropriate informative semantic meaning. The syntax of the language School$_{98}$ is described in section 5.1.

In the following section 5.2 we present in detail the $\pi$-calculus semantics of the language School$_{98}$. The semantics is presented in a form of translations of the syntactic constructs to the $\pi$-calculus process expressions. We keep the translations as local as possible. This requirement of locality complicates the syntax a bit, but it makes the translation clearer.

To make the definition of the language complete we present some built-in classes. The standard data structures will be encoded directly in the $\pi$-calculus and presented in a form of classes. Section 5.3 presents a short illustration of the work required to encode the built-in classes.

This chapter ends with section 5.4, in which we present some programming examples. These examples illustrate the properties of the language School$_{98}$.
5.1 Constructs

In this section, we propose the design of the language School98. We base the design on the ideas presented in Chapter 2. We pick up the discussion about the required properties at the point where the section 2.4 ended. Subsection 5.1.1 describes the properties we want our language to have. We concentrate on the most primitive constructs that are necessary. We leave out many high-level construct described in Chapter 2, but throughout the section 5.2 we point out the possibilities of encoding such construct.

Subsection 5.1.2 presents the syntax of the language School98 together with informative semantics.

5.1.1 Required properties

The object-oriented approach is modern these days. Its popularity comes from the fact that the object-oriented modelling can express the real world phenomena in very natural way. Therefore, it is obvious that we want our language to be object-oriented. We go one step further, and just as in Smalltalk, we try to build a pure object-oriented language. This purity requires every entity in the language to be an object. Even non-object entities must be modelled as objects (see subsection 5.3.1). The difference with Smalltalk is that we do not consider a class as an object. It serves only as a prescription for its instances. A class is a part of the outside world – therefore the creation of new object is realised through a special operator new. The requirement of purity has one major advantage – it offers extra flexibility for the programmer. In addition, the pure object-oriented languages are very robust and expressive. Other good thing about pure object-oriented languages is that they have very simple syntax and semantics, because they include only minimum language constructs. Many language features are represented in form of built-in classes.

We modify the standard object-oriented model in one way. To keep the translation local for every rule allow an object to access only its own private variables. The object cannot access the variables of its ancestor. This extra constrain is only a minor modification and it does not complicate the programmer’s task. In addition, our model with this extra constraint is in some sense purer then for example the Smalltalk object model. In the next chapter, where we introduce a more
complex translation and we exclude this condition. However, for the simplicity of the translations in this chapter, we keep it.

Inspired by the Smalltalk model, we make the language School/98 untyped. In this chapter we design untyped object-oriented language and later in the next chapter we try to extend the language with typing. We leave the typing to the next chapter to be able to concentrate on other important attributes.

So far, the described features can also characterise a sequential language. Now, if we look back to section 2.4, we can find that the concurrent features that look the most basic are data-driven synchronisation and a source of asynchrony in form of asynchronous message passing with the preservation of message order. In addition, every object is equipped with its own processing power, i.e. it can act independently. These features seem to be the most elementary and we think they are sufficient to express many other concurrent features. The expressiveness of the language must also be supported with appropriate library tools that would allow customisation of the basic programming environment to the needs of every application.

The described properties that should our language School/98 satisfy are really only the most fundamental ones. One particular weakness of the language is that it lacks some sophisticated synchronisation mechanisms. Nevertheless, we hope that the specification is open enough to include any further features that we find useful in the future.

We sum the above paragraphs into these points that should describe the main characteristics of the language School/98:

- pure object-oriented language
- instance variables of super class are not accessible to subclasses
- untyped
- data-driven synchronisation
- asynchronous message passing
- active objects
- preserving the order of messages

In the next subsection, we create a language syntax that will include necessary constructs to satisfy the above characteristics.
5.1.2 Syntax

A running program is a collection of interacting objects. At the start of the program, the collection consists only of one root object. One start-up message is send to the root object to initialise the computation. The start-up message must have zero arity. A program environment includes a running program and a collection of classes. New object can be created using the environment service (see expressions below). From the syntactic point of view, a program is definition of program environment, i.e. a collection of classes together with one root object that receives one start-up message.

\[ P ::= C P \]
\[ (\text{new } id) - \text{m()} \]

An object in the language School/98 has its internal state represented by its instance variables and a predefined behaviour. The instance variables and the behaviour – methods, of an object describe its class. Either a class in language School/98 is a base class, or it is derived from some previously defined class. Because a class is not an object, it cannot include any class variables or class methods as Smalltalk class does.

\[ C ::= \text{class } id \text{ vars } V \text{ methods } M \text{ end} \]
\[ \quad \text{class } id \text{ vars } V \text{ from } id \text{ inherit } I \text{ methods } M \text{ end} \]

The inherited methods must be listed to keep the translation rules local. If we employed more sophisticated rules then the \text{inher}it section could be avoided. This will be done in the next chapter. However, for now the \text{inher}it section must contain method names that are inherited.

\[ I ::= id \]
\[ \quad id ; I \]

The \text{vars} section of a class definition defines instance variables. Among the variables included in this section, every object implicitly includes variable me. Variable me contains a reference to the object possessing it. The \text{vars} section has this syntax:
The methods section of a class definition contains methods of an object. A method consists of a method header and a body.

\[
M ::= \ id(id, \ldots, id) = B \\
    \ id(id, \ldots, id) = B \ M
\]

A message body can perform a computation. This computation can change the internal object state. In order to distinguish methods that access state of an object only for reading from those that modify the state we define this syntax:

\[
B ::= \ E \\
    \ A \\
    \ E ; \ A
\]

\[
A ::= \ id ::= E \\
    \ id ::= E ; E \\
    \ id ::= E ; A \\
    \ id ::= E ; E ; A
\]

If a message body is created using the rule \( B \rightarrow E \) then the message body does not directly modify the state. If one of the two remaining rules for non-terminal \( B \) is used then the message body modifies directly the state of an object. Message body constructed using non-terminal \( A \) includes at least one assignment of a new value to an instance variable.

The main part of the syntax is a definition of expressions. Expressions are used in the message body to perform the computation.

\[
E ::= \ E -> id(E, \ldots, E) \quad \text{message-send} \\
    \ \text{new} \ id \quad \text{operator new} \\
    \ id \quad \text{variable} \\
    \ k \quad \text{constant} \\
    \ E ; E \quad \text{sequential composition} \\
    \ E = id ; E \quad \text{seq. composition with binding}
\]
The only expression that can perform a computation is message-send. The message-send construct represents the asynchronous message passing. We do not discuss the synchronous message passing, but it can be easily added to our language. The other syntax construct that can change the computation is the operator `new`. It creates an instance of the class `id`.

Sequential composition evaluates two expressions in sequence, but when the first expression does include message sending, the message is sent asynchronously. This means that the evaluation of the other expression does not have to wait until the result of the first expression is known. It only waits until all the messages, that are to be sent in the first expression are actually sent.

In standard imperative languages, there exists some form of temporary variables or local variables in functions/procedures. In functional languages, there exists some form of let statement that simplifies the programming, and it can be compared to temporary variables of imperative languages – except for an assignment of course. To avoid assignment in the language School/98 and to support some construct similar to let statement in functional languages we added sequential composition with binding. Its natural meaning is evaluate the first expression, bind `id` to the result and then evaluate the second expression.

5.2 Translation

The translation consists of one or two functions per non-terminal (\([\bullet]\), \([\bullet]_c\), \([\bullet]_v(P)\), \([\bullet]_l\), \([\bullet]_t\), \([\bullet]_b(r)\), \([\bullet]_a(r)\), \([\bullet]_v(r)\), \([\bullet]_b, [\bullet]_c(r)\)), a special function `E(\bullet)` for non-terminal `A` and an operator `⊕` together with function `P(\bullet, \bullet)`.

One of the most important parts of the translation is the translation of expressions and we start right with it.

5.2.1 Expression

The question is how the translation function should be defined. What arity should it have and what are the parameters? These questions can easily be answered if we look at how expressions are used. The syntax allows an expression to form a body of a message or a part of a message body. The other thing we must take into account is that methods usually return results. If we go back to subsection 3.3.1 and look at how functions are encoded in the \(\pi\)-calculus, we see
that the function-process receives one special parameter – a result channel. An expression must also know a result channel where it can send the result of its evaluation. We can conclude this paragraph with statement that the translation function for expressions has one parameter – the result channel – and it is denoted as $\tilde{\bullet}_t(c(r))$.

We begin the definition of the translation function $\tilde{\bullet}_t(c(r))$ with the most important form of an expression – message-send. To explore the maximal parallelism we allow all the expressions that form the message parameters to evaluate in parallel.

$$\tilde{E}>m(E_1, \ldots, E_n)_t(c(r)) :=$$
$$\langle v \ rE_1, \ldots, rE_n \rangle \ | \ [E_1]_t(c(r)) \ | \ \ldots \ | \ [E_n]_t(c(r)) \ | \ rE_1?\text{[obj]} \ . \ \text{obj}!\text{[m rA]} \ . \ rA?\text{address} \ . \ rE_1 \ldots rE_n \rangle \ | \ rE_1 \ldots rE_n \rangle$$

$$(\forall i, j \in \{1..n\})(\forall x \neq rE, rE_1)(rE, rE \notin \text{fn}([E_1]_t(c(x)), [E_2]_t(c(x))))$$

In the line 4, the message-send translation reveals the protocol we use in the message passing. We leave this for later and concentrate on the rest of the message-send translation. The results of the parameter evaluation are collected one by one and the method is called supplying it with the result channel.

Now, suppose the expression consists of two expressions connected by sequential composition. The most natural rule for the translation of the sequential composition is simple.

$$\tilde{E}_1 ; \tilde{E}_2)_t(c(r)) := \langle vrE_1 \rangle ([E_1]_t(c(r)) \ | \ rE_1?\text{[va1]} \ . \ [E_2]_t(c(r))$$

$$(\forall x \neq rE_1)(rE_1 \notin \text{fn}([E_1]_t(c(x)), [E_2]_t(c(r))))$$

This translation of the sequential composition shows that the message passing is synchronous – the second expression waits until the first one is completely evaluated. If the first expression is in form of a message-send, then we can see the synchronicity of the message passing. To make the message passing asynchronous we can slightly modify the translation of the sequential composition – the second expression is free to evaluate at any time and the result of the evaluation of the first expression is discarded independently.

$$\tilde{E}_1 ; \tilde{E}_2)_t(c(r)) := \langle vrE_1 \rangle ([E_1]_t(c(r)) \ | \ rE_1?\text{[va1]} \ | \ [E_2]_t(c(r))$$

$$(\forall x \neq rE_1)(rE_1 \notin \text{fn}([E_1]_t(c(x)), [E_2]_t(c(r))))$$
Under this encoding the message passing is completely asynchronous. However, all the sequential properties of the sequential composition are gone. The major problem is that we required the preservation of message order in subsection 5.1.1. Then the sequential composition is the construct that must preserve the order. There are several possible approaches to overcome this problem.

First solution can be like this. We can create a new special rule for the case when the sequential composition includes message-send sub-expressions. The rule would include an acknowledgement, which is signalled after the first message was successfully sent.

\[
\begin{align*}
[E_1->m(E_{i1}, \ldots, E_{in}) \ ; \ E_2]_e(r) & := \\
(\forall \ rE, \ rE_{i1}, \ldots, rE_{in}, \ ack) & \\
(\forall \ [E_1]_e(rE_i) \ | \ [E_{i1}]_e(rE_{i1}) \ | \ \ldots \ | \ [E_{in}]_e(rE_{in}) \ | \\
\ rE_i?\text{[obj]} \ . \ \text{obj!}[m \ rA] \ . \ rA?\text{address} \ . \\
\ address![rE_{i1} \ldots rE_{in} \ rE] \ . \ ack![\]] & \\
\ | \ rE?[\text{val}_1] & \\
\ | \ \text{ack?[]} . \ [E_2]_e(\) & \\
(\forall i, j \in \{1..n\})(\forall x \neq \text{ack}, rE, rE_{i1}) & \\
(\text{ack, } rE, rE_{i1} \not\in \text{fn}([E_{i1}]_e(x), [E]_e(rE), [E_2]_e(x)))
\end{align*}
\]

This rule would satisfy the requirement of the message order preservation, but it is too rigid in the sense that it is suited only for this special problem.

We adopt more general and more flexible solution. We change the whole translation function \([*]_e(r)\). The parameter will no longer be a result channel, but it will be the channel, along which the result channel will come. The reception of the result channel will serve as an acknowledgement that the messages were sent.

The new rule for message passing is now:

\[
\begin{align*}
[E->m(E_1, \ldots, E_n)]_e(r) & := \\
(\forall \ rC, \ rC_{i1}, \ldots, rC_n) & \\
(\forall \ [E]_e(rC) \ | \ [E_{i1}]_e(rC_{i1}) \ | \ \ldots \ | \ [E_n]_e(rC_n) \ | \\
\ rC?[rE] \ . \ rE?[\text{obj}] \ . \ \text{obj!}[m \ rA] \ . \ rA?\text{address} \ . \\
\ rC_{i1}?[rE_{i1}] \ldots rC_n?[rE_n] \ . \ address![rE_{i1} \ldots rE_n \ rE] & \\
(\forall i, j \in \{1..n\})(\forall x \neq rC, rC_{i1})(rC, rC_{i1} \not\in \text{fn}([E_{i1}]_e(x), [E]_e(x)))
\end{align*}
\]

The new rule must take into account the new meaning of the parameter. In the line 4, we collect one by one all the acknowledgements of the message parameter evaluations and send the result channels together with acknowledgement channel \(r\) as parameters. The message body is now independent in handling the parame-
ter result channels. The message body is also responsible for sending the acknowledgement – the receiver can decide the type of message (synchronous/asynchronous) (see section 2.2.2 for asynchronous annotation).

The sequential composition can now be translated in almost the same way as was translated for the first time. We must discard the result only.

\[
[E_1 ; E_2]_e(r) := (\forall rC_2) (E_1]_e(rC_2) \mid rC_1?[rE_1] . (rE_1?[val_1] \\
| [E_2]_e(r)))
\]

\[
rE_1 \not\in fn([E_2]_e(r)) \land (\forall x \neq rC_1)(rC_1 \not\in fn([E_1]_e(x), [E_2]_e(r)))
\]

As next translation rule we discuss the rule for the variable read access. The instance variables in every object will be in form of reference cells. One reference cell for each variable. This directly implies the translation rule.

\[
[id]_e(r) := (\forall rV)(get_{id}! [rV] \mid r! [rV])
\]

To use this rule also for identifiers that bind the results in the sequential composition with binding we create a translation of the named construct with this in mind. The rule is based on the previous rule for simple sequential composition. The difference is in that the value of the previous expression is not discarded, but it is used in the replicated process that simulates a constant variable.

\[
[E_1=id ; E_2]_e(r) := \\
(\forall rC_1) ([E_1]_e(rC_1) \\
| rC_1?[rE_1] . ((\forall get_{id}) \\
| (rE_1?[val_1] . *get_{id}?[r] . r![val_1])) \\
| [E_2]_e(r)))
\]

\[
rE_1 \not\in fn([E_2]_e(r)) \land (\forall x \neq rC_1)(rC_1 \not\in fn([E_1]_e(x), [E_2]_e(r)))
\]

The side condition at the bottom of the rule does not include the channel name get_{id}. This name must be caught in the process [E_1]_e(r) that can include a reference to id and the reference must point precisely at this private name. This translation rule also shows that if several following sequential compositions with binding bind the result to the same identifier only the last one is accessible to the following expressions!

The translation of a constant is simple. The only assumption is that the constant must be one of the built-in constants – see section 5.3.

\[
[k]_e(r) := (\forall rK)(k! [rK] \mid r! [rK])
\]
The translation of the operator \texttt{new} we defer to the subsection 5.2.7.

5.2.2 Method body

The same question about the functions translating a method body must be asked. First, we concentrate on the function for non-terminal \textit{B}. A method should return a result. The body is the place responsible for getting the results. The translation function for the method body must therefore have at least one argument. We can make the parameter an acknowledgement channel, but it is unnecessary and as the discussion about the translation function for non-terminal \textit{A} will show it would complicate the whole translation. Therefore, the parameter supplied by the method translation is the result channel. With this assumption, the translation of the method body without assignment is straightforward.

\[
[E]_B(r) := (\nu r \in C (E)_C (r_C) \mid r_C \in \text{fn}(E) r \in \text{ conclus}(E))
\]

The method body that includes an assignment brings some problems. If the assignment is followed by any expression then it must be clear what value the instance variable represents. Take this fragment of code for example:

\[
\text{...}
\]
\[
x := 10;
\]
\[
\text{print}(x+2);
\]
\[
\text{...}
\]

If the assignment is evaluated strictly – i.e. the next expression is evaluated after the result of the evaluation is assigned to the instance variable – then we lose some concurrency. If the next expression would start its evaluation after the standard acknowledgement then we do not know when the actual assignment happens and the results of the subsequent evaluations would be non-deterministic. We want the evaluation to be deterministic and still not to lose the concurrency features of our language. One possible way is to make the variable temporarily disabled. If we employed this proposed solution, we would allow using the instance variables in almost any trick with assignments. We want to keep the data flow as explicit as possible so we prefer a more functional-like solution.

When objects are modelled in functional languages, every object method must return a new copy of the internal state instead of updating it in place. Despite the
fact that we use updateable data structure to model the internal state, we try to simulate the functional objects. We can achieve similar effect by delaying the assignment to the moment when the whole method body is evaluated and the actual result is known.

Therefore, the part of the code that includes the assignment is transformed to the sequential composition with binding – it is the responsibility of the function $E(\bullet)$. The expressions that follow the assignment can now easily access the new value of the instance variable. The instance variable has the same value throughout the whole method body evaluation and the new value can be accessed using the constants $\text{new}_\text{id}$, where $\text{id}$ is the name of the instance variable that is to be modified.

The process that performs the actual assignments must be inside the scope of identifiers of the form $\text{new}_\text{id}$. One way of achieving this is to modify the translation function for expressions. It would require modification of every rule. To keep the problem of assignments apart from the evaluation of expressions we introduce new operator $\oplus$. This operator takes a set of variables to update and an acknowledgement channel, which will signal the start of the update process. The new operator offers more flexibility and more room for future changes.

The above discussion can be summarised in the next rule, the rule for the translation of the method body with an assignment. The new operator $\oplus$ is used to allow the actual assignments to be performed after the method body is evaluated. The assignments are performed after the signal $\text{start}!\text[end]$. When the assignments are over it is signalled on the channel $\text{end}$.

$$\begin{align*}
[A]_b(r) &:= \\
& (\forall rC, \text{start}, \text{end}) \\
& ([E(A)]_E(rC) \oplus ([A]_A, \text{start}) \\
& | rC?\text{[rE] . rE?[val]} . \text{start}![\text{end}] . \text{end}?[] . r![\text{val}]) \\
& (\forall x \neq rC, \text{start}, \text{end})(rC, \text{start}, \text{end} \notin \text{fn}([E(A)]_E(x)))
\end{align*}$$

The last rule for the translation function $[\bullet]_b(r)$ is just a simple modification of the above rule.
The rules for non-terminal $B$ that include an assignment show that the choice of
the parameter meaning for the translation function $[\star]_B(r)$ (i.e. the parameter $r$ is
a result channel) is reasonable, because the actual result of the evaluation must
be obtained in these rules.

The above rules include three new functions ($E(\bullet)$, $[\bullet]_A$, operator $\otimes$). We explain
them in turn. First is the function $E(\bullet)$, which works over terms build with non-
terminal $A$. It changes the assignments included in the rules for non-terminal $A$ to
the sequential composition with binding.

$$E(id:=E) := E=new_{id} ; Ack$$

$$E(id:=E_1 ; E_2) := E_1=new_{id} ; E_2$$

$$E(id:=E ; A) := E=new_{id} ; E(A)$$

$$E(id:=E_1 ; E_2 ; A) := E_1=new_{id} ; E_2 ; E(A)$$

The only problematic translation is the first one where we used a special constant
Ack as a next expression that determines the result of the whole expression.
Because the result should be an acknowledgement of the assignment, the con-
stant Ack can be the following process:

$$*\text{Ack?}[r] . r![\cdot]$$

Constant Ack will be a built-in constant without a class. More about built-in
classes and constants is in section 5.3.

The translation function for non-terminal $A$ just collects the names of the in-
stance variables that are to be changed after the method terminates.

$$[id:=E]_A := \{id\}$$

$$[id:=E_1 ; E_2]_A := \{id\}$$

$$[id:=E ; A]_A := \{id\} \cup [A]_A$$

$$[id:=E_1 ; E_2 ; A]_A := \{id\} \cup [A]_A$$
What is left is the definition of the operator $\oplus$. The changes performed by this operator must reflect the fact that the assignments must be performed after the evaluation has finished and the processes that accomplish it must be inside the scope of identifiers of the form \texttt{new\_id}. The translation sneaks into the scope of those identifiers by moving itself behind the last expression in the sequence. Therefore, if the expression is built from two sub-expressions in sequence it moves to the second sub-expression.

$$
[[E_1 \; E_2]_e(r) \; \oplus \; (A, a) := \\
  (vrC_1)([[E_1]_e(rC_1) \; | \; rC_1?rE_1] \; . \; (rE_1?[val_1] \\
  \; | \; [E_2]_e(r) \; \oplus \; (A, a))) \\
  rE_1 \not\in \text{fn}([[E_2]_e(r))] \; \land \; (\forall x \neq rC_1)(rC_1 \not\in \text{fn}([[E_1]_e(x), [E_2]_e(r))]
$$

If the structure of the expression is different from the sequential compositions then we know that we are in the last expression in the sequence, if such a sequence exists. Now, the operator $\oplus$ just adds the code for the actual assignments in parallel to the expression translation.

$$
[[E]_e(r) \; \oplus \; (A, a) := [E]_e(r) \; | \; a?[r] \; . \; P(A, r)
$$

The code of the actual assignments – the function $P(\cdot, \cdot)$ – consists of the reception of the signal to start and then follow the assignments. The code of the assignment to a variable \texttt{lg} consists of reading the constant \texttt{QHZBLG} and then setting the variable \texttt{lg} to the received value.

$$
P\{id\} \cup A, r := \\
  (r \text{rcni, rv, ack}) \\
  (\text{get}_{\text{new}_\text{id}}![rv] \; . \; rv?[val] \; . \; \text{set}_id![val \; \text{ack}] \; . \; \text{ack}[] \; . \; P(A, r))
$$

The last action after all the assignments were performed is the acknowledgement for the method body.

$$
P\{\} , r := r[]
$$
The rules for the method body translation are not final and we modify them slightly in the next subsections.

5.2.3 Method

To put the translation of a method into context we first describe how an object will look like in the running system. The message-send translation (see subsection 5.2.1) revealed the protocol we use in the message passing. Now, objects must be constructed in such a way to be able to receive messages using this protocol. Every object is represented by one channel name. At this channel, it waits for requests. The message passing protocol is: the sender asks an object for an actual address of the message named \( m \) and waits for it, then it calls the message located at the received address. On the receiver’s side, an object waits for a request, selects among the method names and returns the address/channel where the appropriate method body is located. The process representing a receiver could look like this:

\[
\text{where the translation of the methods is such that only one method receives the signal and returns its actual address. In order for the object to be able to select one method for evaluation, the method bodies are summed using the summation (also called choice) operator of the } \pi \text{-calculus.}
\]

\[
[i_{d1}, \ldots, i_{dn}] \ B_{\text{m}} := [i_{d1}, \ldots, i_{dn}] \ B_{\text{m}} + [M]_{\text{m}}
\]

One method is translated easily with respect to the other translations. The method must be able to respond to its name with the returning of its address and then it must wait until it is actually called. When the actual call is performed, the method returns an acknowledgement with the result channel. The parameters are translated as constants that become accessible at the time the appropriate parameter is evaluated.
With this translation, the method can be called asynchronously. If the programmer is to have a choice to decide the type of the message (synchronous/asynchronous) then this translation can be easily transformed into such form that the acknowledgement with the result channel is sent after the actual result of the method is known.

### 5.2.4 Method and method body – extended

The translation of the method could be almost complete, but still it does not satisfy all the requirements. The problem not addressed so far is the need to control the evaluation of methods inside one object. The current translation allows any number of methods in one object to be evaluated concurrently. This can lead to errors that are hard to find. The other problem coupled with this one is the preservation of message order.

There are several possible solutions. If we add a counter to an object, we can create a semaphore that can control the exclusive access to the object. The same effect can be achieved if we temporarily disable the reception of new requests. It can be accomplished with this process.

\[ \text{enable} ?[] . \text{obj} ?[\text{name } r] . (\text{name} ![r] \mid [M])_n \]

The reception of messages would be enabled at the time the method finishes its evaluation – this would require only simple modification of \([\text{*}]_n\). This solution is nice and simple but it has one major drawback. When a sender asynchronously sends a message, it is blocked until the receiver is willing to serve it. Such a solution is close to the one proposed in section 2.2.1, where the receiving object had to be locked before any message passing to it. We think that this is too re-
strictive and the amount of concurrency should be greater. The other problem with the above proposition is that there is no guarantee that the sender will ever be allowed to send its message, because under the race-conditions for the re-
ceiver other senders can get to the receiver earlier.

Therefore, we choose to supply every object with a queue of requests – see sub-
section 3.3.6 for the encoding of a queue. The queue guarantees that every sender is served and it stores the messages that are to be served in the order of their arrival. Then the messages are removed from the queue one by one and served.

We can possibly modify the method translation: the called method adds a start-
up channel to the queue and then waits for the actual start-up. After the method body returns the actual result, we would remove the next method from the queue and start it. This looks logical, but it cannot be achieved in this way, because the method body sends the result directly to the sender that called this method and we cannot intercept the communication and catch the result. Therefore, we create a new translation function \([\bullet]_\emptyset(r)\) for the method body. We keep the original translation function for the purposes described in the next subsections. We de-
scribe a way to do build the new function from the old one and show just one example of the new rule. Let the old rule be \([B]_\emptyset(r) := r\). Then the new rule is

\[
[B]_\emptyset(r) := (v\text{start})(\text{put}![\text{start}]. \text{start}?. []). \ R'
\]

\[
\text{start} \notin \text{fn}(R)
\]

where \(R'\) we obtain by exchanging the last sub-process \(r! [\text{val}]\) with the new process \((r! [\text{val}] \mid \text{get}! []).\) For example, the first rule is now:

\[
[E]_\emptyset(r) := (v\text{start})(\text{put}![\text{start}]. \text{start}?. []). \hspace{1cm}
\]

\[
(\forall x \neq \text{start}, rC)(\text{start}, rC \notin \text{fn}(E')(x))
\]

These rules use a slightly modified queue then the one shown in subsection 3.3.6. The difference, as can be seen in the rules, is in that the \text{get} operation does not take the result channel as a parameter. The queue used here must send an acknowledgement along the channel that is at its head. The new queue encoding can be obtained by supplying the following process for the \text{get} operation.

\[
\text{get}?.ulfilled? [c]. c! []
\]
The new queue will be reachable at location queue. Because we created a new function for method body, we must create a new function for methods that uses it. We name it \([\star]_w\) and its rules are the same as the rules for function \([\star]_s\), except that they use function \([\star]_w(r)\) instead of \([\star]_s(r)\).

Now, we execute one method at a time within one object. This policy could possibly be modified to allow more concurrency. One way is to allow the programmer to specify the amount of allowed concurrency (see section 2.2.2). A better solution might provide an automatic decision mechanism. We think that the provided solution with one control queue can be easily extended to allow a more general policy. During the experiments, we designed a solution with two queues that could distinguish between two sets of compatible methods and the solution was just a simple modification of the one proposed here. We do not present it here, because the specification of the language School/98 does not include a mechanism for such a distinction.

5.2.5 Expression - extended

We solved the problem of single-threading an object, but the solution is imperfect because when an object contains a call to itself in any method then the execution of such a call will deadlock.

One approach to solving this problem can be this. Say, if the message-send translation can check if the sender and the receiver are the same, then it can use appropriate protocol and avoid deadlock. On the other hand, if the sender is sent as a message parameter the method translation can check if the sender and the receiver are the same and bypass the queue. These solutions are based on the fact, that we can compare two object i.e. two channel names. We would need a \(\pi\)-calculus mechanism that can compare two channel names and take the appropriate action.

We did not find a suitable solution for comparing two objects so we decided to take a simpler less general approach. We extend the expression syntax category and employ a new rule:

\[
E ::= \ldots \mid \text{self-!m}(E_1, \ldots, E_n)
\]

The syntax of self-message passing is slightly different to indicate that the message is sent synchronously. The new rule assumes the existence of a new object
encoding that includes a separate set of methods for outside senders and for the self-sending. The self-sending set is located at location \texttt{private self} and uses the same protocol as the original set. The original set is used only for the outside senders, now. The self-message-send must be synchronous to keep the object single-treaded, i.e. the method encoding guarantees that the computation continues only after the method returns its actual result.

\[ [\text{self}!m(E_1, \ldots, E_n)]_a(r) := \]
\[
(v \ rM, rC_1, \ldots, rC_n) \]
\[
([E_1]_a(rC_1) \ | \ \ldots \ | \ [E_n]_a(rC_n) \ | \]
\[
\text{private}._a!m[a].rA?address .
\]
\[
rC_1?[rE_1] \ldots rC_n?[rE_n] . \text{address}![rE_1 \ldots rE_n \ rM]
\]
\[
| \ rM?[rE] . rE?[val] . (\text{vret})(\text{ret}![\text{val}] | r![\text{ret}]))
\]
\[
(\forall i, j \in \{1..n\})(\forall x \neq rM, rC_i)(rM, rC_i \notin \text{fn}([E_j]_a(x)))
\]

The proposed solution does not resolve all the possible deadlock situations. If the programmer uses the old self-sending (\texttt{me.m()}) the object would block. The deadlock can also easily happen for example when an object receives a reference to itself from other object and performs a message-send to it. Further investigation would be required to find cleaner solution for this problem.

The same problem is in the case of inheritance when a method of a super class is called. We extend the expression category again and give a new rule.

\[
E ::= \ldots | \text{super}!m(E_1, \ldots, E_n)
\]

The extension exactly follows the previous one. The new translation rule is almost the same as the above one – we only exchange \texttt{private self} with \texttt{private super}.

\subsection*{5.2.6 Variable}

Translation rules for variables are simple if we use the definition of reference cell from subsection 3.3.5. Each variable is represented by one reference cell. The channel received from the reference cell must be accessible for the whole object, i.e. the process representing object must be in scope of these channel names. The translation function \([\bullet]_a\) consequently creates a string of prefixes. These prefixes collect the value of the initial expression and initialise one reference cell. Then these prefixes will be put in front of the object process.
Now the initial expressions must be evaluated separately, the channel names used in their evaluation must be restricted also separately, and all this would be performed by two additional functions. Instead of three functions we can use only one if it take one parameter – a process that will be put as the last process and therefore be inside the scope of all the required channel names.

\[
\text{[id=E; } \nu \text{] } (P) := (v \ rC, r) \\
(\nu \ E)[rC] (rC) \\
| rC?[rE] . rE?[v] . ref![v \ r] . r?[set_id \ get_id] . [V]_v(P)) \\
(\forall x \neq rC, r) (rC, r \notin \text{fn}([E]_e(x), [V]_v(P)))
\]

\[
\text{[id=E] } (P) := (v \ rC, r) \\
(\nu \ E)[rC] (rC) \\
| rC?[rE] . rE?[v] . ref![v \ r] . r?[set_id \ get_id] . P) \\
(\forall x \neq rC, r) (rC, r \notin \text{fn}([E]_e(x), P))
\]

Variables without initialisation are translated as if they were initialised to NULL – special built in constant; see section 5.3.

\[
\text{[id; } \nu \text{] } (P) := [[id=NULL; } \nu \text{] } (P)
\]

\[
\text{[id] } (P) := [[id=NULL] } \nu \text{] } (P)
\]

### 5.2.7 Class

From the translations introduced so far, we can easily deduce how the translation rules for the class look like. An object created from a class must have two sets of methods (subsection 5.2.5) that employ the same protocol shown earlier (section 5.2.1). An object must also include a private queue (section 5.2.4) and instance variables (section 5.2.6). This all can be combined into the following rule:
This rule works fine if we do not include inheritance. The inheritance brings two problems. One problem is that the base class must know the identity of the whole new object not just identity of its private part of the object. The other problem is that the hierarchy of classes must share the request queue. The simplest idea is to remove the queue initialisation from the class process. The class process waits for the queue to be created outside (by operator new). The class process also waits for the identity of the object. This allows the base class to receive the identity of the whole object. The translation of the base class with respect to the above paragraph is:

We show next the definition of a class inheriting from a super class. The new thing lies in the line 6 of the rule where the super class is called to create itself and then after its creation the super class receives the queue and the identity of the whole object – line 7.
The inherited and unchanged methods are added to the method selection. The code for the selection of the inherited methods is created by two new functions each one for one set of method selection.

\[
\begin{align*}
\text{id}\_I := & \text{id}[r] \cdot \text{private}_{super}![\text{id} r] \\
\text{id} \cdot I\_I := & [\text{id}]_I + [I]\_I \\
[\text{id}]^*_I := & \text{id}[r] \cdot \text{super}![\text{id} r] \\
[\text{id}]^*_I := & [\text{id}]_I + [I]^*_I
\end{align*}
\]

If the inherited method is called then the subclass only forwards the name to the super class. To finish the translation we define the rule for the operator `new`. The functionality of this operator was given above and to summarise, it calls the class process and then creates a queue and supplies the new object with the queue and its identity. The last action is to send the new object as the result of the operator `new`.

\[
\text{new id}\_C(r) := (v \text{rq}, r0, ret)
\begin{align*}
 & (\text{id}!\text{[r0]} \cdot \text{r0}[\text{obj private}_{obj} s] \\
 & \text{queue}_{A}!\text{[rq]} \cdot \text{rq}[\text{put get}] \\
 & s!\text{[put get obj private}_{obj}] \cdot \text{ret}!\text{[obj]} \\
 & | \text{r}!\text{[ret]})
\end{align*}
\]

5.2.8 Program

The translation of the whole program is straightforward. If the result returned from the program is required, we can use the parameter `r` as a result channel. The class definitions are translated by their own rules.

\[
[C \ P]^r_C(r) := [C]^C \mid [P]^r^r(r)
\]
CHAPTER 5. LANGUAGE SCHOOL/98

The start-up message is translated as a standard expression evaluation. The result from the evaluation of the start-up message is the result of the whole program and it is returned along the channel \( r \).

\[
\begin{align*}
\text{[(new id).m()]}_r(r) & := (\forall rC)(\text{[(new id).m()]}_r(rC) \\
& \quad | rC? [rE]. rE? [val]. r! [val]) \notag \\
(\forall x \neq rC)(rC \notin \text{fn}([\text{(new id).m()}]_r(x))) \notag
\end{align*}
\]

The translation of a program is almost closed process expression. The only free names in the program are the method names. If we collected all the different method names and restricted the whole program process with these names then we would get a closed process.

5.3 Built-in classes

Standard programming language always comes with the basic library. The basic library is often written in low-level language or it uses low-level virtual machine services as in Smalltalk. We show three examples of standard or so-called built-in classes and constants. The list of built-in features should be longer, but the goal of this thesis is not to give a full description of all the necessary built-in features. The examples given here are included just to give a feeling of the low-level programming necessary to build a complete usable system and to show that some constructs can by created to make the programming easier.

Most of the code of the built-in classes is inspired by the implementation of the basic classes in Smalltalk. However, great caution must be taken in defining methods that serve as program control structures, because in a concurrent system their behaviour is slightly different then in a sequential one.

5.3.1 Null and NULL

**null** is a simple class that includes only one method – **isNull** that returns true.

```ruby
class Null
  methods
  isNull
    true
end
```

The constant **NULL** is an instance of this class. This constant should represent an entity that is not an object.
If we defined the basic class `Object`, then this class would include method `isNull` returning `false`, because its instances do not represent non-object entities.

### 5.3.2 Block

The true low-level programming comes when we try to implement blocks. A block is an expression that can be sent as a parameter without evaluation. Its only method is `value` and this method evaluates the expression. A block can be parameteric and its parameters are channels where the actual values of parameters are. A block can only be created as a constant and the constant is represented as:

```
[:x_1 ... x_n exp]?[r] . (vobj)(r![obj]
 | *obj?[m r] . (m![r] | P))
```

```
P = (vaddress)
    (value?[r] . r![address] . address?[rX_1 ... rX_n r] .
     (rX_1?[val_1] . *get_x_1?[r] . r![val_1]
      | ... | rX_n?[val_n] . *get_x_n?[r] . r![val_n]
      | ![exp]e(r)))
    address ! fn([exp]e(r))
```

### 5.3.3 Boolean

The basic truth-values can be easily encoded is Smalltalk fashion. The constants `true` and `false` are the instances of their appropriate classes. We show just a simple fragment from the class `True`.

```smalltalk
class True
methods
...
&&x
    true
    ifTrueFalse(b_1, b_2)
    b_1.value()
...
end
```
To make to code more readable we define a special transformation that allows a more familiar looking conditional statement. The if statement is a conditional expression followed by two blocks. These blocks are evaluated appropriately by the ifTrueFalse method.

\[
\text{if } \exp b_1 \text{ } b_2 \Rightarrow \exp.\text{ifTrueFalse}(b_1, b_2)
\]

Unfortunately, this simple Smalltalk-like encoding does not work properly in the concurrent setting. The program control structure ifTrueFalse implemented as above does not have the expected properties. Suppose we have this fragment of code:

\[
\text{... if true}
\]
\[
\text{[x.doThis()]
\text{[NULL]; x.doThat();}
\text{...}
\]

The values of block constants are unevaluated expressions and they are sent to the asynchronous message ifTrueFalse. This message returns an acknowledgement, to signal the next expression that it can evaluate, and then it starts to evaluate its own code. This can lead to state when the object denoted by x receives the message doThat before it receives the message doThis. Such behaviour is not what we expect – we expect all messages included in the expression to be sent before the next expression in the sequential composition starts its evaluation. To overcome this problem we can redefine the message ifTrueFalse with synchronous behaviour (see end of subsection 5.2.3). Such encoding is good but it would be too limiting, because every program control structure encoded as a synchronous message would destroy big amount of available concurrency. Therefore, we prefer to provide a solution suited specifically to the needs of every program control structure. We show, for example, how the conditional can be encoded.

The conditional message is encoded in such a way that the next expression that follows the conditional message-send starts to evaluate after the acknowledgement from the appropriate block evaluation is received. With the presented encoding of block evaluation (see subsection 5.3.2), the encoding of conditional
can be easily accomplished by allowing the block evaluation to signal the acknowledgement.

\[
\text{[ifTrueFalse(b_1, b_2)]_e} := \\
(\text{vaddress}) \\
(\text{ifTrueFalse?[r] . r![address] . address?[\text{rB}_1 \ \text{rB}_2 \ r] .} \\
\text{rB}_1?[\text{b}_1] . [\text{b}_1.\text{value()}]_e(r))
\]

This short example of creating a part of the built-in class `True` shows that the encoding of built-in structures must be done with precision. Further effort would be needed to create a full-fledged language.

### 5.4 Examples

Our first example presents one agent from a multi-agent system. This agent pins-up to one environment named `Env1`, it creates a timer and a proxy where it can be reached. When its proxy is triggered it increments variable `x` in the environment. When the timer expires, agent checks if the variable is zero and sets it to 1 if it is zero.

```java
class ExAgent
vars x
from Object inherits isNULL
methods
startUp(manager)
  x:=manager->getEnv('Env1');
  (new Timer)->set(10,me);
  (new Proxy)->set(manager,me)

proxywakeup
  x->get('x') = tmp1;
  tmp1+1 = tmp2;
  x->set('x',tmp2)

timerwakeup
  x->get('x') = tmp1;
  if (tmp1 == 0)
    [1]
    [tmp1] = tmp2;
  x->set('x',tmp2)
end
```
This example shows the basic constructs and the simplicity of the language School\textsuperscript{98}. The basic programming without extensive use of assignments is easy. The second example shows a binary searched tree class only with ability to add a new element. This example illustrates a bit complicated programming style in cases where several interdependent assignments must be performed, caused by the requirement of explicit data-flow. Further refinement of the syntax and the translation could probably lead to a better solution, but it would require deeper analysis of the addressed issues.

```python
class BST
    var key;
    right;
    left
    from Object inherits isNULL
methods
    add(x)
        key := if (key->isNULL)
            [x]
            [key];
        left := if (! (key->isNULL) && (x < key))
            [if (left->isNULL) [new BST] [left] = l;
            l->add(x);
            l]
            [left]
        right := if (! (key->isNULL) && !(x < key))
            [if (right->isNULL) [new BST] [left] = r;
            r->add(x);
            r]
            [right]
end
```
Chapter 6

Language School/98-T

Now, if we take type system described in section 4.2 and we try to assign types to channels in the translation we find that we are not able to do it. The main problem lies in the object-process. An object is identified by one channel and this channel must be able to receive a name of the called method. This name/channel in turn carries a return channel along which an object sends the actual address of the called method. At this address, the method receives the parameters. The type of the method address determines the type of the object identification channel. The main difficulty is in that the methods do not have the same arity. If they had than we would simply use polymorphism to substitute the types of the method parameters. We could possibly use other type system, but it is questionable if a type system that would successfully assign types to our processes exists. We did not concentrate on the search for such type system. We remain with the type system of Turner developed further in the PICT language, because the implementation of the PICT language is available. If the proposed solution is exacted then the PICT implementation offers great opportunity for testing. Even the authors of the PICT language say in [PT97] that PICT “provides a testbed for experiments with statically typed concurrent objects”.

Therefore, in this chapter we modify the translation to the syntax of the PICT language. This modification requires only minor changes to the proposed translation. The more difficult task is to redesign slightly the translation to allow types to be assigned. The addition of the type system makes some changes to the definition of the language inevitable, but we try to keep them to minimal extent. We make the changes as inconsequential as possible, in such a way that the
original untyped program would run in this new environment if it can be correctly typed. We describe the necessary changes in the appropriate subsections (6.2.x) of this chapter. Besides, many of the enhancements included in this chapter can be transferred back to the original language School/98.

Because of the changes in the language definition, we name the new typed language School/98-T. We keep the original language name to indicate that the new language is only an extension of the previous one. The T added at the end of the name means ‘typed’ to signal that this version of our language includes types.

This chapter is organised as follows. Pierce and Turner presented in [PT94] a model of sequential objects is given in $F^{\omega}_{\leq}$, a higher-order explicitly typed $\lambda$-calculus with sub-typing. In [PT95] the same authors pointed out that: “Based on previous experience with foundational models of sequential objects, we believe that any reasonable high-level scheme for concurrent objects with inheritance can be expressed in the setting we have described.” We take the results of [PT94] and transfer them with small modifications into the PICT language. Section 6.1 presents step by step transformation of the results of [PT94]. Then in section 6.2, we merge the sequential objects of section 6.1 with the translation of the language School/98 to obtain the translation of the extended language School/98-T. Together with the new translation we describe the changes in the syntax of the language School/98-T.

## 6.1 Basics

In this section, we show the encoding of structures presented in [PT94] in the similar order as they were introduced in that paper. We also use the same examples that illustrate the encoding. One small difference is that we use objects with updateable state. The reader should be familiar with [PT94] in order to understand this section.

Every object in the proposed model is modelled as an encapsulated state and a set of functions that operate over the state. The representation of the state is hidden to allow only the object methods to access it.

We begin with example of one-dimensional point. The interface type representing the types of the methods for a point is:
Now to make the state accessible only to the methods with the interface type we encapsulate the state and the methods into object using existential type – in the PICT language called package type.

\[ \text{Object} = \text{\{M: (Type -> Type) \{Rep\}[state= \text{Rep} methods= (M \text{Rep})] \}} \]

A point object with interface type PointM has the following type:

\[ \text{Point} = (\text{Object PointM}) \]

We fix the representation of a point to be able to define a class of points to:

\[ \text{type PointR = [x= (Ref \text{Int})]} \]

We also define a constructor that builds the state. This constructor uses the reference cell defined in subsection 4.3.4.

\[ \text{PointState?*[r: PointR] = (new rR: (Ref \text{Int})} \]
\[ \hspace{1cm} \text{(ref![0 rR] | rR?ref}_x = r![x=ref}_x]) \]

Now, the point class with the methods setX – set the x co-ordinate, getX – get the x co-ordinate and bump – move the point by d, represented in PICT is:

\[ \text{PointClass?*[ret: !(PointM PointR)] =} \]
\[ \hspace{1cm} \text{(new gx: ?[PointR \text{!Int}] new sx: ?[PointR \text{Int }![]]} \]
\[ \hspace{1cm} \text{new b: ?[PointR \text{Int }![]]} \]
\[ \hspace{1cm} \text{(ret!sx=sx getx=gx bump=b]} \]
\[ \hspace{1cm} \text{| (gx?*[state x] = state.x.get![x])} \]
\[ \hspace{1cm} \text{| (sx?*[state x] = state.x.set![x r])} \]
\[ \hspace{1cm} \text{| (b?*[state d] =} \]
\[ \hspace{1cm} \hspace{1cm} \text{(new w:^[Int] (state.x.get![w]}} \]
\[ \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{| w?xVal = state.x.set![(xVal+d) r]))}) \]

With this definition of point class, new points object can be created using the operator new defined below. This operator new is different than the keyword new used in the declarations of the new channel name.
As an example, we present a creation of new point object:

```plaintext
val point=new{PointM}{PointR}[PointClass PointState]
```

To continue the description in the same way as in [PT94] we need a way to invoke a method of an object – for example, the `setX` method of point object. We define a special function that performs the actual message `setX` passing.

```plaintext
Point'setX*?[point: (Object PointM)@{Rep}[state methods]
  i: Int
  r: ![]] = methods.setX![state i r]
```

The good thing about the PICT language is that this definition of `Point'setX` is also usable when the subtype of `Point` is supplied as a parameter to this process. To show an example of the object manipulation this next example creates a new point object, then sets a new co-ordinate for it and then gets it back and prints it out. The output of this short example should be 10.

```plaintext
val point=new{PointM}{PointR}[PointClass] in
val []=Point'setX![point 10] in
val x=Point'getX[point] in
println x
```

### 6.1.1 Inheritance

When the inheritance comes to play a new class definition must be used. In the paper [PT94], the extension of a class definition to include reference to object itself was made using the polymorphic fixed-point operator `rec`. The result of the operator `rec` can be simulated in PICT in the following way. The creation procedure will be supplied with one parameter `ret` that is a channel along which the class process sends the methods and a channel `cSelf`, which represents a channel where the actual self-reference comes up. At first, we create new names of channels representing the methods of an object and `cSelf` channel, then return these names and wait for the actual self-reference to come along the `cSelf` channel.
When the self-reference has been received, the methods are ready to be used. The encoding of point class can now be given as:

```hs
PointClass?*[ret: ![[(PointM PointR) !PointM PointR))] =
  (new gx: ?[PointR !Int])
  new sx: ?[PointR Int ![]]
  new b: ?[PointR Int ![]]
  new cSelf: ^[PointM PointR]
  (ret![[setX=sx getX=gX bump=b] cSelf]
    cSelf?self = (cSelf)
    | (gx?*[state r] = state.x.get![r])
    | (sx?*[state x r] = state.x.set![x r])
    | ( b?*[state d r] = (new w:^[Int
class.getX![state w]
    | w?xVal =
      self.setX![state (d+xVal r)])))
```

With this new class definition comes hand in hand the new definition of the `new` operator. It must be changed to accommodate the changes in the class definition. One change is required. Upon the reception of the new object methods and the return channel `cSelf`, it must send the methods back along the `cSelf` channel to allow the new object to be aware of its identity.

```hs
new?*{M}{Rep}{class: ![![{M Rep} !{M Rep}]
  rep: ![!Rep]
  r: ![Object M] =
  (new rc:^[{M Rep} !{M Rep}] new rr: ^Rep
    (class![rc]
     | rc? [objM cSelf]=
       (cSelf?objM | rep![rr]
        | rr? [objR] = r!{Rep}![state=objR methods=objM])))
```

To continue along the lines of [PT94], we define the class `CPoint` that represents a coloured one-dimensional point. This class inherits from its ancestor class `Point` and adds two new methods to manipulate the colour of the point. The new representation must also be created. The new methods and new variables in the state must be added at the end of the record because PICT's type system is sensitive to the order of record fields.

```hs
type CPointR = [x= Int c= Int]```
When this class process is called, it creates the ancestor. Upon the reception of
the ancestor, it sends itself back to the caller and waits for the actual self-
reference to come. When the self-reference is received the class process sends it
to the ancestor and makes the new methods ready.

### 6.1.2 High-level syntax

The above syntax is quite low-level to be usable in real programming. This
subsection exploits the higher-level syntax constructs. This syntax is almost the
same as in [PT94]. The minor extension of the syntax is the introduction of the
sequential composition ($\cdot;\cdot$).

```plaintext
type CPointM = \Rep: Type = [getX= [Rep !Int]
  setX= [Rep Int []]]
  bump= [Rep Int []]
 getC= [Rep !Int]
  setC= [Rep Int []]]

CPointClass?*[ret: ![CPointM CPointR) ![CPointM CPointR)] =
  (new gc: ![CPointR !Int]
  new sc: ![CPointR !Int]]
  new cSelf: !^CPointM CPointR)
  new rSuper: ![CPointM CPointR) ![CPointM CPointR)]
  (PointClass!rSuper)
  | rSuper?[super cSuperSelf]=
    (ret![getX=super.setX
      setX=super.getX
      bump=super.bump
      getC=gc
      setC=sc]
    cSelf]
  | cSelf?self =
    (cSuperSelf!self
     | (gc?[state r] = state.c.get![r])
     | (sc?[state c r] = state.c.set![c r])))
```

When this class process is called, it creates the ancestor. Upon the reception of
the ancestor, it sends itself back to the caller and waits for the actual self-
reference to come. When the self-reference is received the class process sends it
to the ancestor and makes the new methods ready.
We do not explain the translation of this high-level syntax in detail, because it is unimportant for the purposes of this thesis. In addition, the translation presented in the next section is very similar to the translation that could be used here.

6.2 Refinement

The goal of this section is to merge the ideas introduced in the previous section with the translation of the language School/98 to obtain the translation of the language School/98-T. The changes in syntax are introduced in the particular subsections together with their translation.

At the beginning, we take the syntax of the language School/98 and the typed-object encoding shown in the previous section, i.e. the type operator `Object` and the definition of each object in terms of its interface type. Therefore, an object will no longer include variables in its scope as in the translation of the language School/98. We divide each object into the state and the methods. Each method will take the state implicitly as the first parameter just as in the previous section.

So far the general description, and now we describe systematically the necessary changes in syntax and the translation of the language School/98-T to the PICT language. We use a pseudo-functional language in the definitions of the auxiliary functions.
6.2.1 Expression

The translation of the sequential objects to the language PICT allows messages to be sent to an object in a functional way:

\[ \text{Point'setX(point 10)} \]

In our syntax, the same effect has the message-send construction. The translation of the method in School/98 is very similar to the one used in the previous one, but the main difference lies in the new protocol used to send messages. The receiving object must be decomposed using layered pattern and the appropriate method is activated then. The types assigned to each acknowledgement channel \((T, T_{C_1}, \ldots, T_{C_n})\) must be determined by some other function. We do not discuss this problem here, but the types can be inferred from the expressions.

\[
\begin{align*}
[E \Rightarrow m(|T_1, \ldots, T_n|E_1, \ldots, E_n)]e(r) & := \quad : 1 \\
\text{(new } rC: \wedge \wedge \text{Object } T) \text{ new } rC_1: \wedge \wedge T_{C_1} \ldots \text{ new } rC_n: \wedge \wedge T_{C_n} & := \quad : 2 \\
([E]e(r) | [E_1]e(rC_1) | \ldots | [E_n]e(rC_n)) & := \quad : 3 \\
rC?rE = rC_1?rE_1 = \ldots = rC_n?rE_n & := \quad : 4 \\
rE?obj@{Rep}[state methods] & := \quad : 5 \\
\text{methods.m!}\{T_1\} \ldots\{T_n\}[state rE_1 \ldots rE_n r]) & := \quad : 6 \\
(\forall i, j \in \{1..n\})(\forall x \neq rC, rC_1)(rC, rC_i \notin \text{fn}([E_1]e(x), [E_2]e(x))) & := \quad : 7
\end{align*}
\]

The translation of sequence of two expressions stays the same, except the syntax.

\[
\begin{align*}
[E_1 ; E_2]e(r) & := \quad \text{(new } rC_1: \wedge \wedge T_{C_1}) \\
([E_1]e(rC_1) | rC_1?rE_1 = (rE_1?val_1 | [E_2]e(r))) & := \quad \text{fn}([E_2]e(r)) \wedge (\forall x \neq rC, rC_1 \notin \text{fn}([E_1]e(x), [E_2]e(x))) \\
rE_1 \notin \text{fn}([E_2]e(r)) \wedge (\forall x \neq rC_1)(rC_1 \notin \text{fn}([E_1]e(x), [E_2]e(x)))
\end{align*}
\]

The translation of sequence of two expressions with binding is not different either.

\[
\begin{align*}
[E_1=\text{id} ; E_2]e(r) & := \quad \text{(new } rC_1: \wedge \wedge T_{C_1}) \\
([E_1]e(rC_1) | rC_1?[rE_1] = ((\text{new } \text{getid}: ?[!T_{C_1}]) \\
rE_1?val_1 = \text{getid}?[r] = r!\text{val}_1)) & := \quad \text{fn}([E_2]e(r)) \\
rE_1 \notin \text{fn}([E_2]e(r)) \wedge (\forall x \neq rC_1)(rC_1 \notin \text{fn}([E_1]e(x), [E_2]e(x))) \\
rE_1 \notin \text{fn}([E_2]e(r)) \wedge (\forall x \neq rC_1)(rC_1 \notin \text{fn}([E_1]e(x), [E_2]e(x)))
\end{align*}
\]

The first difference is that we have two categories of variables. The variables representing the state form one category and the variables that bind the method parameters together with variables used in the sequential composition with
binding form the other category. For each category, we must define different access protocol. The state variables are reference cells and they can be easily accessed:

\[ [\text{state.x}]_e(r) := (\text{new } rX:\forall T (\text{state.x.get!}[rX] \mid r!rX)) \]

For the second category, we use the original access protocol.

\[ [\text{id}]_e(r) := (\text{new } rV:\forall T) (\text{get}_{id}[rV] \mid r!rV) \]

The translation of constants is much simpler then before, because constants are part of the syntax of values in PICT.

\[ [k]_e(r) := (\text{new } rK:\forall T_k (rK!k \mid r!rK)) \]

Now, we must modify the syntax of expressions to allow the functions already defined in the language PICT to be accessible from our language. We also remove the operator new from the syntax of expressions, because it is available as a function. In addition, the methods defined in super-classes are available as functions. Therefore, we introduce a function application.

\[ E ::= \ldots \mid E(|\text{Type}, \ldots, \text{Type}| E, \ldots, E) \]

The translation of the function application is similar to the message-send – the only difference is that we do not decompose the result of the first expression as an object, but we use the received result as a function name. This change requires slight modifying of lines 5 and 6 in the message-send translation.

We also must add one special variable to the syntax of expressions. Using the current syntax and leaving out the assumption that an object has one variable that serves for the self-reference, (this would require use of recursive types), we add a variable me to the syntax of expressions. This variable can be used only when we want to send the self-reference to other object! This variable may not be used to send self-messages – this would cause a deadlock! The translation of the variable me uses variable self_{outside} that references the methods available for the outside objects (see subsection 6.2.5).

\[ [\text{me}]_e(r) := (\text{new } rO (rO!\{\text{Rep}\}[\text{state self}_{outside}] \mid r!rO)) \]

The use of existential types also suppresses the use of so-called binary methods (methods that include parameters of the same type as the type of the receiver). There are solutions that can overcome the problem of binary methods (see...
[BCC+94]) but none of them is perfect. Therefore, we think that it is no longer the best choice to define all program control structures in terms of methods. Further investigation is required to find out the best representation of program control structures, because in the current setting the only possible choice seems to be to include them among the expressions.

### 6.2.2 Method body

Method body translation requires only minor changes caused by the asynchronous communication used in the language PICT. The changes can be straightforwardly accomplished – in few places where the output prefix is used the process must be divided into two processes in parallel composition to make the output asynchronous. As an example we show one rule – for comparison see subsections 5.2.2 and 5.2.4.

\[
[A]_b(r) :=
\begin{align*}
& (\text{new } rc::^\wedge T_c \text{ new start:!![]} \text{ new end:}^\wedge []) \\
& ([E(A)]_c(rC) \otimes ([A]_a, \text{start}) \\
& | rc?[rE] = rE?[\text{val}] = \text{start}[\text{end}] \\
& | \text{end}[] = (r![\text{val}] | \text{get}[][]) \\
& (\forall x \neq rc, \text{start}, \text{end})(rc, \text{start}, \text{end} \notin \text{fn}([E(A)]_c(x)))
\end{align*}
\]

### 6.2.3 Method

Likewise, method translation requires only small changes caused not only by the asynchronous communication but also by the new protocol used in the message passing.

\[
[id(id_1: T_{i_1}, \ldots, id_n: T_{i_n}):T B M]_m := \\
[[id(id_1: T_{i_1}, \ldots, id_n: T_{i_n}):T B]_m | [M]_m
\]

\[
[id(id_1: T_{i_1}, \ldots, id_n: T_{i_n}):T B]_m :=
\begin{align*}
& (id?^*[\text{state } rE_1 \ldots rE_n r] = \\
& (\text{new } get_{i_{id_1}}[!T_{i_{id_1}}] \ldots \text{ new get}_{i_{id_n}}[!T_{i_{id_n}}] \text{ new return:}^\wedge T) \\
& (r!\text{return} \\
& | rE_i[\text{val}_i = get_{i_{id_i}}^*[r] = r!\text{val}_i) \\
& | \ldots \\
& | rE_n[\text{val}_n = get_{i_{id_n}}^*[r] = r!\text{val}_n) \\
& | [B]_b(\text{return}))
\end{align*}
\]

\[
(\forall i \in \{1..n\})(\forall x \neq rE_i, \text{return})(rE_i, \text{return} \notin \text{fn}([B]_b(x)))
\]
6.2.4 Variable

We must slightly modify the syntax of the variables declaration. We must add
the explicit type annotation. This is necessary to create a state representation
type. The new syntax also requires initialisation of data. This simplifies the
translation a bit and does not complicate the programmer’s task.

\[ V ::= \text{id} : \text{Type} = E \]
\[ \text{id} : \text{Type} = E ; V \]

First, we define three auxiliary functions. Function \( \text{RI} \) takes two lists of names
coupled with type and returns the concatenation of these two lists in format:

\( \text{Label id.} \)

\[ \text{RI} \left( \text{id:T} \right. , \, \text{xs, ys} \) := (id=id) . \text{RI(xs,ys)} \]
\[ \text{RI} \left( \emptyset , \left( \text{id:T} \right. \right. , \, \text{ys \ )} := (id=id) . \text{RI((),ys)} \]
\[ \text{RI} \left( \emptyset , \emptyset \right. := \emptyset \]

Second auxiliary function creates a list of names coupled with type from the
variables-section.

\[ \left[ \text{id:T} = E ; V \right]_{V_{\text{IN}}} := (\text{id:T}) \cdot [V]_{V_{\text{IN}}} \]
\[ \left[ \text{id:T} = E \right]_{V_{\text{IN}}} := (\text{id:T}) \cdot \emptyset \]

The last auxiliary function creates a process that evaluates the initialisation ex-
pression. The returned value is then used as an initial value in the creation of a
new reference cell. After the reference cell is created the next variable can con-
tinue or the process supplied as a parameter can continue.

\[ \left[ \text{id:T} = E ; V \right]_{\text{x}(P)} := \]
\begin{align*}
\text{(new } rC: & \, T_{E} \text{ new } r:(\text{Ref } T) \left( [E]_{E} (rC) \\
& | \, rC?rE = rE?val = \text{ref}{\!}\{T\}[val \ r] \\
& | \, r?id = [V]_{x}(P)) \right) \\
(\forall x \neq rC, r \notin \text{fn}([E]_{E}(x), P)) \\end{align*}

\[ \left[ \text{id:T} = E \right]_{\text{x}(P)} := \]
\begin{align*}
\text{(new } rC: & \, T_{E} \text{ new } r:(\text{Ref } T) \left( [E]_{E} (rC) \\
& | \, rC?rE = rE?val = \text{ref}{\!}\{T\}[val \ r] \\
& | \, r?id = P)) \right) \\
(\forall x \neq rC, r \notin \text{fn}([E]_{E}(x), P)) \\end{align*}
With the help of the above auxiliary function, the translation of the variables-section is simple. First the new variables are created and then the state of the ancestor is created. The new state is created from the old state merged with the new variables. Expression $\lambda_\alpha(id)\in S$ means the list of state variable names that are in the set $S$ paired with the name $id$.

$$[\lambda]_\alpha(s, id) :=
\begin{cases}
\text{new } r \in \{\text{idState}'*r\} = \\
[\lambda]_\alpha(\text{idState}'*r) = \\
\{r\}$[RI(\lambda_\alpha(id), ,())] = r[RI(\lambda_\alpha(id), ,[\lambda]_\alpha())])}
\end{cases}$$

The case when the class is a base class is even simpler. We just leave out the section where the ancestor is called.

$$[\lambda]_\alpha(s, \emptyset) :=
\begin{cases}
\text{idState}'*r = \\
[\lambda]_\alpha(r)[RI((),,[\lambda]_\alpha())])
\end{cases}$$

### 6.2.5 Class

The main syntax change we introduce in language School/98-T is that we leave out the inherit section of the class declaration. This syntax is closer to standard object-oriented languages, but it makes the translation a bit complex. This change is also forced by the fact that in the language PICT the sub-typing rules for records require the records to have the same order of common fields. We could not keep the original syntax, because the method redefinition would be quite complicated for the programmer and it would make the code of the class unclear and almost unreadable.

Before we define the actual class translation, we define auxiliary functions. First function $\lambda_\alpha$ takes two lists of names. The first list of names represents a list of methods defined in the super-class. The second list of names represents methods defined in the sub-class. The result of this function is a list of elements of form Label id that will be used as a list of functions valid for the sub-class. The function definition is simple. If the first list is not empty yet then it checks if the considered element is also in the second list and if it is then the name from the second one is used – the method is redefined in the sub-class. If the considered name is not in the second list then the old name (super . x) is used instead – the
method is not redefined. If the first list is empty then all the new names are added – list of newly defined methods.

\[
\text{MI( (id:T) . xs, M ) := (id=id) . MI(xs, M-(id:T)) <- Member((id:T), M)}
\]

\[
\text{MI( (id:T) . xs, M ) := (id=super.id) . MI(xs, M)}
\]

\[
\text{MI( (), (id:T) . M ) := (id=id) . MI((), M)}
\]

\[
\text{MI( (), () ) := ()}
\]

The second auxiliary method just collects the names of the newly defined and the redefined methods in the class.

\[
\begin{align*}
[id(id_1: T_{id_1}, ..., id_n: T_{id_n}): T \ B \ M]_{\text{ON}} := \\
(id:?[\text{idR} \ ?T_{id_1} ... ?T_{id_n} !T]) . [M]_{\text{ON}}
\end{align*}
\]

\[
\begin{align*}
[id(id_1: T_{id_1}, ..., id_n: T_{id_n}): T \ B]_{\text{ON}} := \\
(id:?[\text{idR} \ ?T_{id_1} ... ?T_{id_n} !T]) . ()
\end{align*}
\]

The third function creates a list of declarations from a set of names coupled with type. These declarations will be used in the class translation to restrict the names of the defined methods.

\[
\text{N((id:T) . M) := (new id:T) . (new id_{\text{private}}:T) . N(M)}
\]

\[
\text{N(()) := ()}
\]

Now we can easily define class translation. The translation function takes one parameter – a set of previously defined classes. When a class is a base class, then this parameter can be ignored. We slightly modify the form of class process – see examples in section 6.1 – to make the distinction between the methods that are called from outside of an object and those that are called through the self-reference. Therefore, the class process includes two sets of methods just as in the translation of the language School\textsubscript{98} – see subsection 5.2.7. The class process relies on the operator \texttt{new} to use the correct protocol in the object creation.
The auxiliary functions are used in line 5 – local declarations of new methods and line 6 – the class process returns two records that are created from the lists of the outside and the private method names.

When we define a sub-class translation, we modify the already shown School\textsubscript{98} class process in the same way as we did in the above translation of the base class. The new point in the translation is in line 9 – we use the parameter of the translation function to get the list of methods defined in the super-class. The expression \(S_c(id)\) means the list of method names that are in the set \(S\) paired with the name \(id\).

Coupled with the above encoding of class definition process is again the operator \texttt{new} which creates a new instance of a class. This operator is a minor modification of the one shown in subsection 6.1.1. The difference lies in the fact that we use two sets of methods and the private methods should not be accessible from outside the object. The private methods are bound to the wildcard pattern in line 9 to show that they are not used anywhere.
6.2.6 Program

The translation of the whole program needs the extract six components from each translated class: representation type, interface type, the representation definition process, the class process, the list of instance variables and the list of methods. Therefore, we define new function \( \text{\textbullet}_{\text{\textbullet}}^6 \) that takes the set of previously defined classes and creates the required components.

\[
\begin{align*}
\text{class id} & \text{ vars V methods } M_{\text{\textbullet}}^6(S) := \\
\text{type} \ 'idR' & = [RX(),[V]_{\text{\textbullet}}^6] \\
\text{type} \ 'idM' & = \text{\textbullet}\text{\textbullet} \text{Rep}:\text{Type} = [MX(),[M]_{\text{\textbullet}}^6] \\
[V]_{\text{\textbullet}}^6(S,\langle\text{empty}\rangle) & \\
\text{class id} & \text{ vars V methods } M_{\text{\textbullet}}^C(S) \\
RI() & ,[V]_{\text{\textbullet}}^6 \\
MI() & ,[M]_{\text{\textbullet}}^6 \\
\end{align*}
\]

\[
\begin{align*}
\text{class id from id}_5 & \text{ vars V methods } M_{\text{\textbullet}}^6(S) := \\
\text{type} \ 'idR' & = [RX(S_5(id)_5),[V]_{\text{\textbullet}}^6] \\
\text{type} \ 'idM' & = \text{\textbullet}\text{\textbullet} \text{Rep}:\text{Type} = [MX(S_5(id)_5),[M]_{\text{\textbullet}}^6] \\
[V]_{\text{\textbullet}}^6(S,id)_5 & \\
\text{class id from id}_5 & \text{ vars V methods } M_{\text{\textbullet}}^C(S) \\
RI(S_5(id)_5) & ,[V]_{\text{\textbullet}}^6 \\
MI(S_5(id)_5) & ,[M]_{\text{\textbullet}}^6 \\
\end{align*}
\]

In the above function definition, we used two new auxiliary functions \( MX \) and \( RX \). These functions take lists of methods respectively instance variables that are defined in the ancestor and in the class under translation. They act just as functions \( MI \) and \( RI \) and the only difference is the form of the returned list elements. Functions \( MX \) and \( RX \) return lists with element in the form \( Label \ Type \).
Now, the translation of the whole program is straightforward. The translation scans the code and creates classes one-by-one and the appropriate list of instance variables and methods is always added to the set of previously defined classes. In the last line, the name of the class is extracted from the class definition.

\[
[C \ P]_\rho(\text{Struct}) := \\
\text{let } (\text{typeR t}yp\text{c procR procC S}_R \ S_C) = [C]_{\alpha\xi}(\text{Struct}) \text{ in} \\
(\text{typeR t}yp\text{c procR procC | procC | } [P]_\rho(\text{Struct } \cup \{\text{id } S_R \ S_C\})))
\]
Chapter 7

Conclusions and further work

As stated in the introduction chapter, we tried to design a simple and flexible concurrent object-oriented language. The language School\textsubscript{98} is based on standard Smalltalk-like object-model with concurrent extensions. The object model includes only active objects – each object possesses its own processing power. The model does not include classes as objects. A class is just a template for its instances. This template exists outside the object world. Each object can request new instance of any class by using the operator \texttt{new}. Each object can send a message to itself or other object or it can change its internal state. Message passing, new object creation, sequential composition and assignment to instance variables are the only syntactic constructs that can control the execution in the language School\textsubscript{98}. Other program control structures are given in form of methods. These methods are significantly different then their counterparts in Smalltalk. The concurrent environment requires great caution in the implementation of these methods – as section 5.3 showed. Many built-in classes must be implemented specifically and directly in the \(\pi\)-calculus.

The concurrent features in the language School\textsubscript{98} are present in form of asynchronous message passing, data-driven synchronisation and active objects. These features are integral part of the language specification. They guided the creation of the mobile calculus semantics.

The extension of the proposed language – School\textsubscript{98}-T – added not only the type system, but changed the design of the whole language. The used type system forced us to include the program control structures in the syntax. In addition, the type system and the \texttt{PICT} language forced us to create more complex translation.
The translation of the whole program and class definition had to be changed significantly, because the PICT language requires precise ordering of the record fields. Other translation functions remained very similar; the type system did not affect them very much.

We can conclude that on the general level we fulfilled the main goal of this diploma thesis. On the other side, in some parts of this thesis we already pointed out that the proposed solution is not perfect and in many details the subject must be studied further. In the remainder of this chapter, we discuss the good points and the points that need further investigation.

The major weakness of the language School$_{98}$ is the lacks of a sophisticated synchronisation mechanism. The problem with synchronisation policies is that we find many proposed techniques unsatisfactory. Therefore, we included only the most natural one – data-driven synchronisation. We think that it is necessary to develop new concurrent constructs that can make the programmer’s work easier. We hope that the proposed language has open enough specification to incorporate possible future enhancements – either in form of new synchronisation techniques or in form of other possible improvements.

One possible enhancement can, for example, be done in the School$_{98}$-T language. The PICT language offers simple form of pattern matching and therefore the patterns could be possibly used in the method header instead of variables. The pattern matching could then be possibly extended to a more complete form. Other simple enhancement can be the yield command to allow other messages to be received while the currently evaluated one is interrupted. We listed only two enhancements, but there are many different ways how to extend our simple language.

The second example in section 5.4 points at one difficult problem – assignment. We tried to avoid it as much as possible, but because we encode the state as an updateable data structure we must allow the assignment to instance variables. Our programming language makes the complicated conditional assignments very painful. By keeping the assignment only at the top level of the method body we made clear that only one assignment to each instance variable is possible. Further effort should be put here to create more refined form of assignments to offer nicer and less complicated programming style.
Because a program in language School\textsubscript{98} can fail – see subsection 3.2.2 for the definition of failure – we designed language School\textsubscript{98}-T. The typing removed many failure cases. Nevertheless, the translation of the language School\textsubscript{98}-T is tricky and only the translation rules guarantee the avoidance of failure. The weak point in the translation is the rule for the operator \texttt{new} and the class definition. School\textsubscript{98}-T typing does not provide clean distinction between the private and the outside methods. The distinction is visible only in the treatment of each set of methods in the translation, but once the original program is translated, the code in the PICT language does not provide enough information to check its correctness. Further investigation is required to solve this problem in cleaner and safer way.

The goal of this thesis was to provide theoretical design of a concurrent object-oriented language. Nevertheless, we encoded the language School\textsubscript{98}-T in the PICT language and this leaves big opportunity for possible implementation. The further work might be to refine the proposed solution to the form that can be directly usable in the PICT language. Then the electronically available version of the PICT language can be used for the implementation.

The current implementation of the language PICT is sequential. Despite the fact that its authors work on the distributed version, the new implementation of the π-calculus could be a challenge. An interesting future project might be to implement a concurrent object-oriented language on top of an existing concurrent object-oriented operating system; for example OS COOL described in [LJP93].

To finally conclude this thesis, we can say that we created a simple, flexible, robust and transparent programming environment that can hopefully serve as a good starting point for future projects.
Bibliography


[Ame87] Pierre America. POOL-T: A Parallel Object-Oriented Language. In [YT87].


[YST+87] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada and Yasuaki Honda. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In [YT87].
