Implementing the $n$-calculus in the VeriCode Proof Tool

Erik Angelin

Master of Science Thesis
Stockholm, Sweden 2006
Implementing the π-calculus in the VeriCode Proof Tool

ERIK ANGELIN

Master’s Thesis in Computer Science (20 credits) at the IT Program
Royal Institute of Technology year 2006
Supervisor at CSC was Mads Dam
Examiner was Johan Håstad

TRITA-CSC-E 2006:133
ISRN-KTH/CSC/E--06/133--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication
KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.csc.kth.se
Abstract

We present an implementation of the process algebra $\pi$-calculus in the proof assistant VCPT, which supports reasoning in first order $\mu$-calculus, an extension of first order logic with fixed point operators. Our goal is to determine if we can use VCPT to support reasoning about behavioural specifications of processes. The specifications will be given as formulas in the logic. We represent processes as terms, using natural numbers for names, and provide definitions for the various syntactical operations and the transition relation. Example proofs are given to demonstrate practical use of the implementation. Suggested future work is to improve the performance of the implementation, extend the implementation with further concepts from $\pi$-calculus theory, and to evaluate other approaches to implementing the $\pi$-calculus in the assistant.

Referat

En implementation av $\pi$-calculus i bevisassistenten VCPT

Vi presenterar en implementation av processalgebran $\pi$-calculus i bevisassistenten VCPT. Assistenten stöder resonerande i en logik kallad första ordningens $\mu$-calculus. Denna logik är en variant av första ordningens logik med fixpunktoperatorer. Målet med arbetet är att utvärdera om VCPT och nämnd logik är användbara för att resoner kring specifikationer av processers beteende. Specifikationerna ges som formler från logiken. Processer representeras som en lämplig typ av term, där vi använder heltal för att representera namn (kanaler och meddelanden). Nödvändiga syntaktiska operationer och den centrala transitionsrelationen definieras som fixpunktsformler samt lemma. Ett par exempelbevis presenteras för att illustrera praktisk användning av implementationen. Möjliga forslutningar på arbetet är att förbättra implementeringsprestanda, att implementera ytterligare begrepp från teorin kring $\pi$-calculus, samt att utvärdera alternativa sätt att implementera $\pi$-calculus i assistenten.
Acknowledgements

I would like to thank my supervisor Mads Dam for letting me explore the related subjects in my own pace, generously tutoring me through the parts I did not initially grasp, and keeping me on track whenever I digressed into irrelevancies.

An earlier master’s student of Mads’, Mikael Lagerkvist, deserves a special thanks for helping me get the proof assistant up and running and sharing the word of mouth and hard lessons learned on how to use it.

I also owe a debt of gratitude to the rest of the formal methods crowd I have got to know during this project: Dilian Gurov, Irem Aktug, Mika Cohen, Wen Xu and my opponent Karl Palmskog have all contributed in various ways with interesting conversations and enthusiasm.

Finally, I would like to thank my girlfriend Mia for just about everything else.
## Contents

1 Introduction .......................................................... 1  
   1.1 The role of this thesis .............................................. 3  
   1.2 Thesis structure .................................................. 3  

2 Background material .................................................. 5  
   2.1 $\pi$-calculus .................................................... 5  
       2.1.1 Informal introduction ....................................... 5  
       2.1.2 Formal definitions .......................................... 9  
   2.2 First order $\mu$-calculus ....................................... 13  
       2.2.1 Syntax ...................................................... 14  
       2.2.2 Semantics and related vocabulary ......................... 16  
       2.2.3 Introduction to the proof system ......................... 18  
   2.3 The VeriCode Proof Tool ......................................... 21  
       2.3.1 Constructing proofs ........................................ 21  
       2.3.2 Terms and formulas ........................................ 23  
       2.3.3 Configuration ............................................... 24  
       2.3.4 Example proof session ..................................... 24  

3 Implementation of the $\pi$-calculus ................................. 27  
   3.1 Process terms ................................................... 27  
   3.2 Syntactical definitions ......................................... 28  
       3.2.1 Names of process terms and actions ....................... 28  
       3.2.2 Alpha equivalence .......................................... 29  
       3.2.3 Capture-avoiding substitution .............................. 29  
   3.3 The transition relation .......................................... 31  
       3.3.1 L-ALPHA ................................................... 31  
       3.3.2 Tactics .................................................... 32  

4 Rationale and alternatives .......................................... 35  
   4.1 The shape of process terms ...................................... 35  
   4.2 The representation of names ..................................... 36  
   4.3 Capture-avoiding substitution .................................. 37  
   4.4 The transition relation .......................................... 38
4.4.1 Treating replication ........................................... 38
4.4.2 Treating implicit alpha conversions ...................... 38

5 Proofs using the implementation .......................... 41
  5.1 A simple specification ...................................... 41
  5.2 A transition relation lemma ................................. 42
  5.3 Infinite shout ................................................ 44

6 Conclusion and future work ................................. 49
  6.1 Future work with the implementation .................... 49
    6.1.1 Improving performance ................................. 50
    6.1.2 Extending the theory ................................ 51
    6.1.3 Re-implementing ...................................... 51
  6.2 Future work with the proof assistant ................... 51
    6.2.1 Documentation ...................................... 51
    6.2.2 A tool for tactic drafting ............................ 52
    6.2.3 Review of substitution code ......................... 52
    6.2.4 Correcting possible bugs ............................. 52

References ....................................................... 53

Bibliography ...................................................... 53

A MPW-logic ...................................................... 55
  A.1 Overview .................................................. 55
  A.2 Implementation .......................................... 57

B Fixed point theory ............................................ 59

C Many-sorted first order logic ............................... 63
  C.1 Syntax ..................................................... 63
  C.2 Semantics ................................................ 64

D First order μ-calculus ....................................... 67
  D.1 Semantics ................................................ 67
  D.2 A proof system ......................................... 68

E Implementation excerpts .................................... 73
  E.1 Set ........................................................ 73
  E.2 Function ................................................. 74
  E.3 π-calculus ............................................... 76

F A bug in VCPT ............................................... 79
Chapter 1

Introduction

Computers turn up most everywhere: in dvd-players, watches, dishwashers, cellphones, mp3-players and obviously on our desks. In certain areas we place a lot of trust in them. Whenever we fly an aircraft, do online banking, or subject ourselves to medical radiation treatment, we depend on computers and in particular the software governing their behaviour, to perform the job correctly.

There are several popular methods for increasing the quality and trustworthiness of software, for instance technical reviews or audits, pair programming and developer testing. While these methods are often effective and adequate they have a common weakness. Typically they are informal, meaning people study code or specification, try to simulate in their heads or on a piece of paper how the system will behave, and then judge whether this behaviour is correct or not. For testing, the choice of what to test and with what input is decided similarly. Depending on the experience of people involved, this may not be sufficient.

Formal methods is about applying rigorous mathematical and logical methods to system development, specification and verification to ensure the resulting system behaves correctly. These methods may be too cumbersome for general projects, but are increasingly applied in areas where the costs of failure are unacceptable.

This thesis is related to the field of formal methods known as process algebra. Process algebras are useful for reasoning about systems whose main characteristic is how they communicate or otherwise interact with other systems, the common examples being distributed systems, (secure) communication protocols and object oriented systems. In contrast, they may not be suitable as a reasoning tool for more computational qualities of systems, such as proving some algorithm always produces the correct result.

Intuitively a process algebra is an unambiguous language we use to create a description of a system, a model. The system may have an implementation, in which case the model can be an abstraction for studying how the system communicates. Alternatively, the model can constitute a prototype for a system to be built provided the model appears to behave correctly. The language generally has constructions for the sending and receiving of messages across some notion of a channel, and various control structures for sequential, conditional or parallel execution.
CHAPTER 1. INTRODUCTION

A model roughly describes the overall state of a system just before the system is started. To mimic the change of state in a system as it runs and performs actions, a process algebra comes with a set of transition rules. These allow us to deduce, for a given model of a state, what actions can be performed, and what the corresponding model will be for the system state after the state change. In effect, by using the rules we can perform a stepwise symbolic execution of a model. By considering all transition rules, we cover all possible interactions that can occur in a particular state.

In this thesis we use the process algebra $\pi$-calculus. This is an example of a value-passing process algebra, meaning that the messages that are sent carry a value to the receiving end. As we will see later on, the values are fundamentally also communication channels. This may seem to be a limitation, but it is in fact possible to represent values from an arbitrary data type in the $\pi$-calculus. The models we create with the $\pi$-calculus are referred to as processes. One distinguishing feature of the $\pi$-calculus is that the language includes an operation for creating private communication channels. These channels can only be used by the process creating them, and by any process to which the private channel has been sent. This feature is useful for modeling the establishment of a session between two parties. A client may for instance contact a server on a public well known channel, establish a private communication channel, and then continue communication over the new channel.

Having a model of our system, we would generally like to know whether it satisfies some specification of its intended behaviour. To be of any use this specification must be stated somehow in terms of the process algebra. This can be achieved in a number of ways. It is common to give the specification as a simpler model which is known to have the intended behaviour. For instance, the specification model $S$ may not use processes running in parallel, while the system model $P$ does. The goal then becomes to prove that the two models behave essentially the same, meaning that an external system cannot tell the difference between $S$ and $P$ merely by communicating with them.

An alternative, which we are interested in here, is to give the specification as a formula from a modal logic. These logics have constructions that refer to the various actions a model can perform. A natural language interpretation of such a specification could be that if the server model receives a request across a particular channel, it will always eventually send a response across the same channel. Another example could be that no communication can bring the system to an unexpected halt. The goal is to prove that the model satisfies the specification using some proof system.

For specifications we use a first order logic with fixed points $^1$. While this logic does not have any direct constructions for referring to actions of models, we can achieve the same effect using an appropriately defined relation. As an alternative specification logic we provide a translation from a variation of a modal logic presented in [MPW93].

To prove that a model satisfies a specification given as a logic formula, we must handle the various logical connectives and, where appropriate, perform symbolic execution of the model to determine what actions might or must be performed. This work is tedious and error-prone to perform by hand. To ease the burden one can make use of a proof assistant. A proof assistant is a program that ensures correctness of all steps in a proof and which can

---

$^1$First order $\mu$-calculus
1.1. THE ROLE OF THIS THESIS

Automate recurring tasks.

Generic proof assistants are not targeted to reasoning about a particular subject. Instead, they support the construction of proofs in some logic using a proof system. Provided the logic is sufficiently expressive, we can formulate known facts about the subject, and questions whose truth we would like to decide, as formulas. Then, provided the proof system is sufficiently complete, we may be able to deduce that the question, or perhaps its negation, follows from the facts. There is a general trade-off between having an expressive logic and a complete proof system.

For this thesis we will be using the proof assistant VCPT, which supports the previously mentioned fixed point logic. VCPT has previously been used to study Java Card, Erlang [FGN+], and most recently, Java bytecode [Lag].

1.1 The role of this thesis

Our goal is to determine if we can use VCPT for reasoning about behavioural specifications of $\pi$-calculus processes, given as formulas from some specification logic, and whether we can make it reasonably convenient. More concretely, we provide an embedding of the syntax and transition rules of the $\pi$-calculus in the first order $\mu$-calculus of the tool, and program helper functions for elaborating proofs.

The $\pi$-calculus has previously only been implemented in more established proof assistants based on more expressive logics. If we can achieve roughly the same level of functionality using VCPT, this would give an indication of the usefulness of the tool and related logic. A secondary contribution is to learn the $\mu$-calculus, proof theory in general and the use of proof assistants.

We will not consider the general problem of extracting a model from a system, neither the formulation of a specification as a logic formula.

One central subject of process algebras is proving two models behave equally, mentioned in the introduction. We generally say that two models behaving equally are bisimilar. In this project we will not explicitly deal with proving models bisimilar. However it may be possible to formulate a particular equality in first order $\mu$-calculus, and in that case the machinery developed could come in handy.

1.2 Thesis structure

We will begin with some background material on the $\pi$-calculus, the first order $\mu$-calculus and the proof assistant. Then we will present key parts of our implementation of the basic $\pi$-calculus theory, followed by motivations for our choices and a presentation of alternatives. To illustrate working with the implementation we give a few example proofs. Finally, we briefly discuss our results and suggest future work.
Chapter 2

Background material

In the following sections we present necessary material to understand the parts of the implementation. First we give a short introduction to the $\pi$-calculus. We will only cover those parts we are interested in implementing. Then, we briefly present the first order $\mu$-calculus, which is the logic used in VCPT. For the reader interested in a deeper understanding, we refer to the appendix. After the section on $\mu$-calculus we turn to its treatment in VCPT and general usage of the proof assistant.

2.1 $\pi$-calculus

This material is condensed from [Mil99, SW01, MPW89], with most definitions from [SW01].

2.1.1 Informal introduction

We can think of a process algebra as a unambiguous language for modeling systems. A $\pi$-calculus model of a system is referred to as a process. A process can perform actions, and by doing so evolve to another process reflecting the state change in the system. If a process $P$ can perform the action $a$ to become the process $P'$ we call this a transition, and write as follows.

$$ P \xrightarrow{a} P' $$

The process $P'$ of a transition is sometimes referred to as the derivative of the transition.

We begin our presentation by a series of examples. Suppose we are modelling a once-only beverage vending machine.

$$ \text{slot(coin).} \{ \text{coin} = \text{penny} \} \backslash \text{spout tea.0} $$

The symbols slot, coin, penny, spout and tea are called names. Names can generally be understood as communication channels, but are also used to represent simple pieces of data. Names can be sent between processes, and compared to other names. We sometimes refer to a name as a channel when it is primarily used as such.
Returning to our beverage machine, the so called input prefix \( \text{slot(coin)} \) means the process can first receive a name, sent from another process, through a channel \( \text{slot} \) (representing the coin slot). The name \( \text{coin} \) acts as a placeholder, which will be replaced by whatever name was received. Provided the name \( \text{penny} \) was received, verified by the match prefix \([\text{coin} = \text{penny}]\), the name \( \text{tea} \) can be sent through the \( \text{spout} \) channel using the output prefix \( \text{spout tea} \). Finally, the process becomes the inactive process \( 0 \).

We can write the first transition of the process as follows.

**Example 2.1.1 (Initial beverage machine transition)**

\[
\text{slot(coin)[coin = penny]} \xrightarrow{\text{spout tea} 0} \text{spout tea} 0 \\
\text{spout tea} 0 \\
\text{spout tea} 0
\]

This is called a bound input action, which is one half of an interaction between two processes. The sending and receiving of names in the \( \pi \)-calculus is synchronous. For two processes to exchange a name, one must perform a bound input, and the other a simultaneous output action on the same channel.

Suppose we introduce a user, which inserts a penny through the coin slot, accepts whatever comes from the spout, and then pours it into the sink. The initial transition of such a process would be as follows.

**Example 2.1.2 (Initial user transition)**

\[
\text{slot penny} \xrightarrow{\text{spout (beverage).sink beverage} 0} \text{spout (beverage).sink beverage} 0 \\
\text{spout (beverage).sink beverage} 0
\]

This is an example of a free output action, representing the sending of a name \( \text{penny} \) through a channel \( \text{slot} \). Let us now consider the possible initial transitions these processes can perform when running in parallel, indicated by the \( | \) symbol.

**Example 2.1.3 (Initial user and beverage machine transition)**

\[
\text{slot(coin)[coin = penny]} \xrightarrow{\text{spout tea} 0} \text{spout tea} 0 \\
\text{spout tea} 0 \\
\text{spout tea} 0 \\
\text{spout tea} 0 \\
\text{spout tea} 0
\]

The action \( \tau \) is the so called silent action which represents any action occurring internally in a process, unavailable for external interaction. As is seen, the placeholder \( \text{coin} \) for the received name has now been replaced by the sent \( \text{penny} \).

The parallel composition of processes does not necessarily force the subprocesses to interact. The subprocesses can still perform their actions separately, but then on behalf of the complete process. There are some conditions for this which we return to later on.

**Example 2.1.4 (Beverage machine subprocess performs transition)**

\[
\text{slot(coin)[coin = penny]} \xrightarrow{\text{spout tea} 0} \text{spout tea} 0 \\
\text{spout tea} 0 \\
\text{spout tea} 0 \\
\text{spout tea} 0 \\
\text{spout tea} 0
\]
2.1. \(\pi\)-CALCULUS

Here, the beverage machine subprocess performs its bound input action. Similarly, the user subprocess could have performed its free output action. Some external process could have interacted with the complete process by performing a suitable complementary action, something we will not illustrate here.

If we let the user and beverage machine processes perform a sequence of transitions, the result could be as follows.

**Example 2.1.5 (Possible sequence of user and machine transitions)**

\[
\text{slot}(\text{coin}), [\text{coin} = \text{penny}] \text{spout tea}.0 \ \xrightarrow{\tau} \ \text{spout}(\text{beverage}).\text{sink} \text{beverage}.0
\]

\[
[\text{penny} = \text{penny}] \text{spout tea}.0 \ \xrightarrow{\tau} \ \text{spout}(\text{beverage}).\text{sink} \text{beverage}.0
\]

\[
0 \ \xrightarrow{\text{sink tea}} \ 0 \ \xrightarrow{\text{sink tea}} \ 0 \ | \ 0
\]

Had the user process sent, say, the name \textit{pound}, the beverage machine could not have performed any further actions as a result of the match prefix \([\text{pound} = \text{penny}].

If we want to model the fact that preparing tea involves some internal action taken by the machine, we could use a tau prefix \(\tau\) as follows.

\[
\text{slot}(\text{coin}), [\text{coin} = \text{penny}] \tau \text{spout tea}.0
\]

After receiving a penny, the machine could perform the following transition.

**Example 2.1.6 (Beverage machine performing an internal action)**

\[
[\text{penny} = \text{penny}] \tau \text{spout tea}.0 \ \xrightarrow{\tau} \ \text{spout tea}.0
\]

This is the same type of internal action as occurred earlier when the machine and user interacted, only now it originates from an explicit prefix.

Suppose the beverage machine allows the user to choose between tea and coffee via a selector. We could then write the corresponding process using the choice construction \(+\), as follows.

\[
\text{slot}(\text{coin}), [\text{coin} = \text{penny}] . \text{selector}(\text{beverage}).
\]

\[
((\text{beverage} = \text{tea}) \text{spout tea}.0 + [\text{beverage} = \text{coffee}] \text{spout coffee}.0)
\]

If coffee was chosen, the machine process could perform the following transition.

**Example 2.1.7 (Beverage machine with coffee transition)**

\[
([\text{coffee} = \text{tea}] \text{spout tea}.0 + [\text{coffee} = \text{coffee}] \text{spout coffee}.0)
\]

\[
\text{spout coffee} \ \xrightarrow{\tau} \ 0
\]
Suppose we have a somewhat simplified beverage machine which can deliver tea multiple times instead of just once. We could then write the corresponding process using the replication construction \(!\), and a user process as follows. (The equational definitions of \(Machine\) and \(User\) are not part of the \(\pi\)-calculus but merely used to save space)

\[
Machine \triangleq \!\select\text{(beverage)}.[\text{beverage} = \text{tea}]\spout \text{tea}.0
\]

\[
User \triangleq \select \text{tea}\spout (\text{beverage}).0
\]

The first transition of the machine and user in parallel could be as follows.

**Example 2.1.8 (Replicated beverage machine transition)**

\[
User | Machine \xrightarrow{\tau} \spout (\text{beverage}).0 | [\text{tea} = \text{tea}]\spout \text{tea}.0 | Machine
\]

Here, the user process has performed a free output action to select tea. Meanwhile, the replicated machine process has created a new parallel process, an instance, to handle the request for tea from the user. This instance has performed a bound input action to accept the tea selection, and is now ready to serve tea.

Note that the replicated machine process remains, and can handle further requests. To be strict, if several parallel user processes were introduced and caused the creation of several machine instances, there would be no sense of identity of these instances. It is not necessarily the instance whose button was pushed that will serve tea to that particular user. To guarantee such behavior we must make use of another facility of the \(\pi\)-calculus.

To present this, we leave the world of beverage machines and consider a common client-server scenario. We often want the same server process to handle a whole session with a client. For instance, the server process may keep some contextual information: what has happened so far, the identity of the client etc.

We can model this in the \(\pi\)-calculus by creating one server process per client (using the \(!\) construction), and ensuring only that particular client can communicate with its server process. As all communication takes place along names, this can be achieved by establishing a private communication channel between them.

Below follows a client process which creates a new private name, and then sends it to a server process to establish contact. All further communication uses this name. In our example the client queries the server for the current date. (\(C\) represents some suitable continuation)

**Example 2.1.9 (Client sending new session channel)**

\[
\nu \text{session}(server, session, session, date, C). server(session) \xrightarrow{\text{session date, C}}
\]

Here, the client process has performed a new type of action, a bound output action.

A server would receive this new name using the normal input prefix. We introduce the following short-hand for the server process. (\(S\) similarly represents some suitable continuation)
2.1. $\pi$-CALCULUS

\[
P ::= M \\
| \ P \ | \ P \quad \text{Parallel composition} \\
| \nu z(P) \quad \text{New} \\
| \ !P \quad \text{Replication} \\
\]

\[
M ::= 0 \quad \text{Inaction} \\
| \pi.P \quad \text{Sequencing} \\
| \ M + \ M \quad \text{Choice} \\
\]

\[
\pi ::= \tau \quad \text{Tau prefix} \\
| \ a \ b \quad \text{Output prefix} \\
| \ a(y) \quad \text{Input prefix} \\
| \ [a = b]\pi \quad \text{Match prefix} \\
\]

Figure 2.1. Syntax of the $\pi$-calculus, with names of the constructions

\[\text{Server} \triangleq !\text{server}(s).s(query).[query = date]S\]

The following example illustrates the establishment of the session channel.

**Example 2.1.10 (Client and server establishing session channel)**

\[
\nu\text{session}(\text{server session.session date}.C) \mid \text{Server} \xrightarrow{\tau} \nu\text{session}(\text{session date}.C \mid \text{session(query)}.[query = date]S \mid \text{Server})
\]

Here, the client process has performed a bound output as presented earlier. The server process has forked an instance which has received the private communication channel and is ready to handle requests from the client process. The server process can fork of further instances to handle other clients, and each client and server instance pair would communicate using their own private channel.

2.1.2 Formal definitions

With an intuitive understanding of processes and transitions, we now turn to the definitions on which we base our implementation. This includes the complete grammar of $\pi$-calculus, reproduced in fig.2.1, and a proper definition of the transition relation $\rightarrow$ which lets us determine what transitions are possible for a given process.

The symbol $P$ of fig.2.1 represents process and is what we have seen examples of earlier.

To define the transition relation, we first need a few syntactical definitions concerning the use of names in processes.

Both the input prefix $a(y)$ and the $\nu y$ construction are called binders and are said to bind the name in the position indicated by $y$. In the process $a(y).P$, the name $y$ becomes bound with the process $P$ as scope, and similarly for the process $\nu y(P)$. Any occurrence of
a name which is within scope of a binder for that name, is a bound occurrence, and refers
to the name in the binder. If there is no binder in scope, the occurrence is said to be free.
We denote by $bn(P)$ the set of names with a bound occurrence in $P$, and by $fn(P)$ the set
of names with a free occurrence, as illustrated in example 2.1.11.

Example 2.1.11 (Free and bound names of processes)
In the process $P$ below, the set of free names (names with a free occurrence) is \{a, b, c, q\}
and the set of bound names is \{b\}. There is no relation between the bound $b$ and the free
$b$. The scope of the bound $b$ is as indicated.

$$P \triangleq (a(b). \overline{b}q.0) \mid \tau b.0$$

In for instance example 2.1.3, we displayed a transition where an input prefix place-
holder (a bound name), was replaced by the name received. This replacement is performed
by applying a substitution. To apply a substitution $fn = y$ to a process $P$, as in $Pfn = y$,
means replacing each free occurrence of $y$ in $P$ with $n$, provided this does not lead to unintended
capture of $n$ by a binder. This is illustrated in example 2.1.12.

Example 2.1.12 (Applying a substitution)
Here we substitute penny for coin in a familiar subprocess.

$$(\text{coin} = \text{penny})\overline{\text{spout tea}.0}\{\text{penny/coin}\} =$$

$$(\text{penny} = \text{penny})\overline{\text{spout tea}.0}$$

Suppose instead we would like the machine to supply tea after the insertion of a coin,
not a penny. To achieve this we attempt to substitute coin for penny in the following process.

$$\text{slot(coin)}\{\text{coin} = \text{penny}\overline{\text{spout tea}.0}$$

The problem is that simply replacing penny with coin would make the new occurrence of
coin bound to whatever was input through the slot, as follows.

$$\text{slot(coin)}\{\text{coin} = \text{coin}\overline{\text{spout tea}.0}$$

This is not what we want, since tea will be output whatever we insert. The problem is known
as an unintended binding. To avoid it, we rename the bound coin as part of the substitution.
The result could be as follows.

$$(\text{slot(coin)}\{\text{coin} = \text{penny}\overline{\text{spout tea}.0})\{\text{coin/penny}\} =$$

$$\text{slot(payment)}\{\text{payment} = \text{coin}\overline{\text{spout tea}.0}$$

One note to make here is that the particular name used in a binder is irrelevant for
the overall behaviour of a process. Either the name is just a placeholder for a name to
be received. Or, some arbitrary new name used only by the processes which share it as a
private channel. Processes that vary only in the choice of bound names are said to be alpha
equivalent, see def.1.
2.1. $\pi$-CALCULUS

Definition 1 (Alpha equivalence)
To alpha convert a process is to replace some subprocess $\nu z(Q)$ with $\nu w(Q[w/z])$, provided $w$ does not occur in $Q$, or similarly for $a(z).Q$. Two processes are alpha equivalent, $P=_{\alpha}Q$, if one can be transformed to the other by some number of successive alpha conversions.

Example 2.1.13 (Alpha equivalent processes)
The process $Q$ below is alpha equivalent to $P$ in example 2.1.11, $Q=_{\alpha}P$. The set of free names is still \{a, b, c, q\} and the set of bound names is \{z\}.

\[
Q = a(z).z.q.0 | \bar{z}.b.0
\]

The transition relation can now be defined by a set of proof rules, which allow us deduce exactly when a process can perform a particular transition. These rules are reproduced in fig.2.3.

The names of the transition rules are prefixed by “L-” for late, referring to when in the deduction of an interaction the input prefix placeholder is replaced by the received name. There is also an early set of rules, which we will not consider in this thesis. The suffix “-R” is short for for left. Symmetrical “-R” rules have been omitted. Some of the rules refer to the names occurring in actions, explained in fig.2.2.

To deduce whether a process can perform a transition, we match the process with the conclusion of the proof rules, and then attempt to deduce the transitions and verify the name conditions in the premises of the rule. This continues until we reach one of the proof rules without further premises.

Some transitions are deduced in example 2.1.14.

\[
\begin{array}{|c|c|c|}
\hline
\alpha & n(\alpha) & bn(\alpha) \\
\hline
\bar{a}b & \{a, b\} & \emptyset \\
\hline
a(b) & \{a, b\} & \{b\} \\
\hline
\bar{a}(b) & \{a, b\} & \{b\} \\
\hline
\bar{\tau} & \emptyset & \emptyset \\
\hline
\end{array}
\]

Figure 2.2. Names and bound names of actions

Example 2.1.14 (Deducing previously presented transitions)
The following transition from example 2.1.1 was deduced using the L-INP transition rule.

\[
\text{slot}(\text{coin}).[\text{coin} = \text{penny}]\text{spout tea}.0 \xrightarrow{\text{slot}(\text{coin})} [\text{coin} = \text{penny}]\text{spout tea}.0
\]

The following transition from example 2.1.2 was deduced using the L-OUT transition rule.

\[
\text{slot penny}.\text{spout(\text{beverage})}.\text{sink \text{beverage}}.0 \xrightarrow{\text{slot penny}} \text{spout(\text{beverage})}.\text{sink \text{beverage}}.0
\]
The following transition from example 2.1.3 was deduced using \( \text{L-COMM-R} \), where for the user process we used \( \text{L-OUT} \), and for the machine process we used \( \text{L-INP} \).

\[
\text{slot(coin).}[\text{coin} = \text{penny}] \Downarrow \text{spout tea.0} \uparrow \text{slot penny.spout(beverage).sink beverage.0} \xrightarrow{\tau} [\text{penny} = \text{penny}] \Downarrow \text{spout tea.0} \uparrow \text{spout(beverage).sink beverage.0}
\]

The following transition from example 2.1.4 was deduced using \( \text{L-PAR-L} \) and \( \text{L-INP} \). The condition on names succeeds because the bound name coin of the action does not occur freely in the user process.

\[
\text{slot(coin).}[\text{coin} = \text{penny}] \Downarrow \text{spout tea.0} \uparrow \text{slot penny.spout(beverage).sink beverage.0} \xrightarrow{\tau} [\text{penny} = \text{penny}] \Downarrow \text{spout tea.0} \uparrow \text{spout(beverage).sink beverage.0}
\]

The following transition from example 2.1.6 was deduced using \( \text{L-MATCH} \) and \( \text{L-TAU} \).

\[
[\text{penny} = \text{penny}] \uparrow \text{spout tea.0} \xrightarrow{\tau} \text{spout tea.0}
\]

The following transition from example 2.1.7 was deduced using \( \text{L-SUM-R} \), \( \text{L-MATCH} \) and \( \text{L-OUT} \).

\[
([\text{coffee} = \text{tea}] \Downarrow \text{spout coffee.0} + [\text{coffee} = \text{coffee}] \Downarrow \text{spout coffee.0}) \xrightarrow{\text{spout coffee}} 0
\]
2.2 FIRST ORDER $\mu$-CALCULUS

$$L\text{-}\text{ALPHA}: \frac{P =_\alpha Q}{P \xrightarrow{\alpha} P'} Q \xrightarrow{\alpha} P'$$

Figure 2.4. Alpha equivalence transition rule

The following transition from example 2.1.9 was deduced using $L\text{-OPEN}$ and $L\text{-OUT}$. The condition on names succeeds because the names server and session differ.

$$\nu \text{ session } (\text{server session.session date.C}) \xrightarrow{\text{server(session)}} \text{session date.C}$$

It is common in presentations of the $\pi$-calculus to avoid the issue of actual bound names by working “up to” alpha equivalence. This effectively means allowing alpha conversion of a process at any time. We have seen this in example 2.1.12 where we had to perform an alpha conversion to avoid unintended capture.

As suggested in [SW01], when deriving transitions, this “up to” amounts to having an extra transition rule presented in fig.2.4. This rule is necessary for instance to ensure bound names agree in the $L\text{-CLOSE}^*$ transition rules, and is illustrated in example 2.1.15.

Example 2.1.15 (Transition involving the $L\text{-ALPHA}$ rule)
The transition below, from example 2.1.10, was deduced using $L\text{-CLOSE-L}$, where for the user process we used $L\text{-OPEN}$ and $L\text{-OUT}$. For the server process, we first fork of an instance using $L\text{-REP-ACT}$. To this instance we then have to apply $L\text{-ALPHA}$ to convert the name $s$ to session, after which we used $L\text{-INP}$. For reference, $\text{Server}$ was defined as $s(\text{query.}[\text{query = date}])S$.

$$\nu \text{ session } (\text{server session.session date.C}) \mid \text{Server} \xrightarrow{\text{}} \nu \text{ session } (\text{session date.C} \mid \text{session(query.}[\text{query = date}])S \mid \text{Server})$$

2.2 First order $\mu$-calculus

The $\mu$-calculus$^1$ can be seen as an extension of first order logic with operators for least and greatest fixed points.

The proof assistant we use supports reasoning in this logic. The principal description is [Fre] p. 15-28, and this section is largely a summary.

We first give the syntax of the logic, followed by examples for an intuitive understanding of the semantics of formulas, and finally we give an introduction to a proof system.

For a summary on first order logic and fixed points, we refer to the appendix.

$^1$We drop the “first order” qualification
CHAPTER 2. BACKGROUND MATERIAL

2.2.1 Syntax

Just as for the first order logic, part of the syntax is determined by a signature defining the
terms of the current subject. If we are reasoning about the natural numbers, we might use
the signatures in example 2.2.1 or 2.2.2. For our π-calculus reasoning later on, we will use
a signature similar to example 2.2.3 for the language of processes.

Example 2.2.1 (Signature for the natural numbers)

sort NAT \triangleq zero
| succ : NAT

The term succ(succ(zero())) would represent the natural number two.

Example 2.2.2 (Signature for sets of natural numbers)

sort SET \triangleq emptyset
| pair : NAT, SET

The term pair(1, pair(2, emptyset)) represents the set containing the natural numbers 1
and 2.

Example 2.2.3 (Signature for process terms of the π-calculus)

sort PROCESS \triangleq summation : SUM
| compose : PROCESS, PROCESS
| new : NAME, PROCESS
| replication : PROCESS

sort SUM \triangleq stop
| sequence : PREFIX, PROCESS
| choice : SUM, SUM

sort PREFIX \triangleq tau
| out : NAME, NAME
| in : NAME, NAME
| match : NAME, NAME, PREFIX

We have left out the definition for the term sort sort NAME, but suppose it was defined
to be any character from the alphabet. The process 0 | a b.0 would then be represented as follows

\text{compose}(\text{sum}(\text{stop}()), \text{sum}((\text{sequence}(\text{out}(a, b)), \text{sum}(\text{stop}()))))

The basic syntax of μ-calculus is given in fig.2.5, and abbreviations are given in fig.2.6.
Here, sort x is some sort from the signature. The t, are terms formed according to the
signature, or a term variable of appropriate sort. One such term variable is x occurring in
the Π formulas.

Formulas of the shape λx : sort x.ϕ are called abstractions. These are meant to be ap-
plied to some term t of sort sort x, as in (λx : sort x.ϕ) t.

Formulas of the shape μU : sϕ.ϕ are least fixed point formulas, which allows recursion.
Complementary greatest fixed point formulas of the shape νU : sϕ.ϕ can be found among
the abbreviations. These formulas are also typically applied to some term. We will see
2.2. FIRST ORDER $\mu$-CALCULUS

$$\phi ::= t_1 = t_2$$

$$| \phi_1 \lor \phi_2$$

$$| \neg(\phi)$$

$$| \exists x : \text{sort}_x.\phi$$

$$| \lambda x : \text{sort}_x.\phi$$

$$| (\phi) t$$

$$| \mu U : s_\phi.\phi$$

$$| (\mu U : s_\phi.\phi)^\beta$$

$$| U$$

Figure 2.5. Syntax of the $\mu$-calculus

\[
\begin{align*}
t_1 \neq t_2 & \triangleq \neg(t_1 = t_2) \\
\text{true} & \triangleq \phi_1 \lor \neg(\phi_2) \\
\text{false} & \triangleq \neg(\text{true}) \\
\phi_1 \land \phi_2 & \triangleq \neg((\phi_1) \lor \neg(\phi_2)) \\
\phi_1 \implies \phi_2 & \triangleq \neg(\phi_1) \lor \phi_2 \\
\forall x : \text{sort}_x.\phi & \triangleq \neg(\exists x : \text{sort}_x.\neg(\phi)) \\
vU : s_\phi.\phi & \triangleq \neg(\mu U : s_\phi.\neg(\phi[\neg(U)/U])) \\
t : \phi & \triangleq (\phi) t \\
\lambda x_1 : \text{sort}_1, \ldots, x_n : \text{sort}_n.\phi & \triangleq \lambda x_1 : \text{sort}_1, \ldots, \lambda x_n : \text{sort}_n.\phi \\
\exists x_1 : \text{sort}_1, \ldots, x_n : \text{sort}_n.\phi & \triangleq \exists x_1 : \text{sort}_1, \ldots, \exists x_n : \text{sort}_n.\phi
\end{align*}
\]

Figure 2.6. $\mu$-calculus formula abbreviations

examples of this later on. The variation with a $\beta$ only occurs in the proof system, and will not be used here.

Not all formulas we can construct by following the grammar are proper formulas, for instance $(\text{true}) t$ does not make sense. By giving each formula a formula type describing what sort of abstraction the formula is (not an abstraction at all, or how many and what kinds of term arguments it takes), we restrict ourselves to the formulas which are well typed. In short, the following rules must be followed.

- The equality symbol can only be used with terms of the same sort
- We must use an appropriately sorted number of terms as arguments for abstractions and the fixed point operators
- For the fixed point formulas the inner formula $\phi$ must be of the same sort (typically an abstraction) as the fixed point formula itself
- Any occurrence of the fixed point predicate variable $U$ in the inner formula must be under an even number of negations
2.2.2 Semantics and related vocabulary

The semantics of formulas (found in section D.1), defining what formulas are true or false, can be disheartening unless you are familiar with fixed points. For an intuitive understanding we give the following examples and two often cited slogans about fixed points:

- Least fixed points means finite looping.
- Greatest fixed points means (possibly) infinite looping.

Example 2.2.4 (The first three primes)
Assume we are using the earlier signature for the natural numbers, but writing with our more familiar symbols for convenience. Then we can define the property of being one of the first three primes with the following abstraction formula.

\[ \text{FirstPrimes} \overset{\triangleq}{=} \]
\[ \lambda P : \text{nat}. P = 2 \lor P = 3 \lor P = 5 \]

Note that FirstPrimes is only a short-hand for the formula. Now, for instance the formula \((\text{FirstPrimes}) 3\) would be true.

Example 2.2.5 (The even numbers) We can define the property of being an even number with the following least fixed point formula.

\[ \text{Even} \overset{\triangleq}{=} \]
\[ \mu E : \text{nat} \rightarrow \text{prop}. \]
\[ \lambda N : \text{nat}. \]
\[ N = \text{zero}() \lor \]
\[ \exists N' : \text{nat} . N = \text{succ} (\text{succ} (N')) \land (E) N' \]

Now, the following formulas would be true.

- \((\text{Even}) 2\)
- \((\neg(\text{Even}) 3)\)

\((\text{Even}) 1\) on the other hand, would be false.

Example 2.2.6 (Set membership)
Assume our signature contains the earlier definition of sets of natural numbers. Then we can define set membership as follows.

\[ \text{Member} \overset{\triangleq}{=} \]
\[ \mu M : \text{nat} \rightarrow \text{set} \rightarrow \text{prop}. \]
\[ \lambda N : \text{nat} . M S : \text{set}. \]
\[ \forall N' : \text{nat} . \exists S' : \text{set}. \]
\[ S = \text{pair} (N', S') \land \]
\[ (N = N' \lor ((M) N) S') \]

Now, the following formulas would be true.
2.2. FIRST ORDER μ-CALCULUS

• ((Member) 1) pair(0, pair(1, emptyset))
• ¬(((Member) 2) pair(0, pair(1, emptyset)))

Example 2.2.7 (Function evaluation)
Assume our signature contains the following definition of a function over naturals.

sort FUN ≡ undefined
| compose : NAT, NAT, FUN

The term \( \text{compose}(1, 2, F) \) is intended to represent a function that evaluates 1 to 2, and any other arguments according to the continuation function \( F \).

We can define function evaluation as follows.

\[
\text{Eval} ≡ \\
\mu E : \text{fun} → \text{nat} → \text{nat} → \text{prop}.
\lambda F : \text{fun} \cdot \lambda X : \text{nat} \cdot \lambda Y : \text{nat}.
\exists F' : \text{fun} \cdot \exists X' : \text{nat} \cdot \exists Y' : \text{nat}.
F = \text{compose}(X', Y', F') \land \\
(X = X' \land Y = Y' \lor \\
X \neq X' \land (((E) F') X) Y)
\]

Now,

\[
(((\text{Eval} \ \text{compose}(1, 3, \text{compose}(1, 4, \text{undefined})))) \ 1) \ 3
\]
would be true, but

\[
(((\text{Eval} \ \text{compose}(1, 3, \text{compose}(1, 4, \text{undefined})))) \ 1) \ 4
\]
false, since the 1 to 3 association takes precedence in our definition.

Example 2.2.8 (Specification)
Assume our signature defines a process sort as in example 2.2.3, and an action sort as follows.

sort ACTION ≡ freeout : NAME, NAME
| boundout : NAME, NAME
| boundin : NAME, NAME
| tau

Assume also that \( \phi \) is some fixed point formula such that \(((\phi) P) A) P' \) holds exactly when we can deduce using the transition rules that the process \( P \) can perform the action \( A \) and then continue as the process \( P' \).

Then the following formula expresses that a process can perform an infinite number of consecutive silent (\( \tau \)) transitions.

\[
\text{EternalInternal} ≡ \\
\nu EI : \text{process} → \text{prop}.
\lambda P : \text{process}.
\exists P' : \text{process}.
(((\phi) P) \text{tau}()) P' \land (EI) P'
\]
CHAPTER 2. BACKGROUND MATERIAL

Now,

\[(\text{EternalInternal}) \text{ replication}(\text{summation}(\text{sequence}(\text{tau}()), \text{summation}(\text{stop}()))))\]

would be true, but

\[(\text{EternalInternal}) \text{ summation}(\text{sequence}(\text{tau}()), \text{summation}(\text{stop}()))\]

false, since the process can only perform one silent transition.

While the semantics allows us to judge the truth of a formula, it is not a very practical method. In the next section we introduce a proof system, which gives syntactical criteria for truth. This approach is what we use in proof assistants. To properly explain the proof system, we must first introduce the notion of semantical consequence.

Say \( \Gamma \) is a set of formulas, and \( \varphi \) is a single formula. If whenever all formulas of \( \Gamma \) are true, \( \varphi \) is also true, then we say \( \varphi \) is a semantical consequence of \( \Gamma \) and write \( \Gamma \models \varphi \). We also require that any variables occurring freely on both sides are interpreted the same in all places. The formulas of \( \Gamma \) are separated by a comma, which can be read as “and”. Some semantical consequences are given in example 2.2.9.

**Example 2.2.9 (Some semantical consequences)**

Remember \( N \) and \( S \) refer to the same number and set respectively on the left and right hand side.

\[
\begin{align*}
\text{false} & \models \\
& \models ((\text{Member}) 1) \text{ pair}(0, \text{pair}(1, \text{emptyset})) \\
((\text{Member}) N) S & \models \forall E : \text{nat}.((\text{Member}) N) \text{ pair}(E, S) \quad \dagger \\
(\text{FirstPrimes}) N, (\text{Even}) N & \models N = 2 \\
((\text{Member}) 3) \text{ pair}(1, \text{emptyset}) & \models 47 = 11 \\
\dagger & \text{ Note the similarity to an earlier example. }
\end{align*}
\]

We also allow several formulas on the right hand side, but in that case we only require one of these formulas to be true whenever all formulas on the left hand side are true. Commas used to separate the formulas on the right hand side can be read as “or”. For an example, see 2.2.10.

**Example 2.2.10 (Semantical consequence with multiple formulas on l.h.s)**

Given suitable definitions of the relations \( \text{lessThan} \) and \( \text{greaterThan} \), the following would be a semantical consequence.

\[(\text{lessThan}) N) 3, ((\text{greaterThan}) N) 0 \models N=1, N=2\]

**2.2.3 Introduction to the proof system**

The whole proof system can be found in the appendix, section D.2. Here we give a shorter presentation, and introduce some concepts.

We use a proof system to completely syntactically determine whether a given semantical consequence holds. The proof system consists of a set of proof rules and axioms,
with which we can construct proofs. If we can construct a proof of a particular semantical
consequence, then we can be certain the semantical consequence holds.

The proof system is based on sequents, \( \Gamma \vdash \Delta \). Here, \( \Gamma \) and \( \Delta \) are multisets of formulas,
meaning the formulas may occur multiple times in the sets. We say a sequent \( \Gamma \vdash \Delta \) is valid
if it truly is a semantical consequence (\( \models \)).

Each proof rule is a schema consisting of a conclusion sequent and a set of premise se-
quents, presented as in fig. 2.7. For every instantiated proof rule, when all premise sequents
are valid, the conclusion sequent must also be valid. To understand the rationale for a proof
rule, it may help to read a sequent \( \Gamma \vdash \Delta \) as an implication between a conjunction and a
disjunction, as follows.

\[
\bigwedge_{\gamma \in \Gamma} \gamma \quad \Rightarrow \quad \bigvee_{\delta \in \Delta} \delta
\]

Axioms are given as proof rules without premises.

**Example 2.2.11 (Trivial proof)**

\[
\begin{align*}
0 & = 1 + 0 = 1 \quad \text{ID} \\
\vdash 0 = 1 & \quad \Rightarrow \quad 0 = 1 \quad \Rightarrow \quad R
\end{align*}
\]

When attempting to prove some particular semantical consequence \( \Gamma \models \Delta \) holds, we
start out with the initial goal sequent \( \Gamma \vdash \Delta \).

We then read the proof rules bottom up and attempt to match the goal sequent with
the conclusion sequent of each proof rule. There may be several proof rules applicable,
depending on the formulas occurring in the goal sequent, but we only choose one at a time.
If, for instance we have a goal sequent with an implication formula on the right hand side,
we can apply the proof rule in fig. 2.8, as we have done in the example proof 2.2.11.

Then we attempt to prove each of the premise sequents of the matched proof rule similarly until we can establish each premise sequent is valid.
CHAPTER 2. BACKGROUND MATERIAL

This procedure successively reduces our initial goal sequent into simpler subgoals, building a tree of sequents. There are primarily four ways to establish that a goal sequent, a leaf, is valid. First, it could be an instance of an axiom, such as g.2.9, giving no further premises to prove. Second, it could be an instance of a lemma. A lemma is some sequent we have previously proven, or otherwise trust, to be valid. Third, by looking at the complete proof tree we may realise that we have already proven the sequent valid on another branch, in which case we can subsume the current goal, referring to the other branch. Finally, by looking at the proof constructed so far we may realise that a variant of this sequent occurs earlier in the proof and has been regenerated because we are processing a fixed point formula. If certain conditions concerning approximation ordinals hold, we may discharge the goal. This will occur in proofs in a later section.

Example 2.2.12 (There is a number different from zero and one)
Below follows a simple proof that there is a natural number different from both zero and one. The signature from example 2.2.1 is used for the representation of numbers. Formulated as a semantical consequence, this would be

\[
\vdash \exists N : \text{nat} \neq \text{zero}() \land N \neq \text{succ}(\text{zero}()).
\]

\[
\frac{
\text{succ}(\text{succ}(\text{zero}())) = \text{zero}() \vdash \\
\text{succ}(\text{succ}(\text{zero}())) \neq \text{zero}() \vdash \\
\text{succ}(\text{succ}(\text{zero}())) \neq \text{succ}(\text{zero}())
}{
\vdash \exists N : \text{nat} \neq \text{zero}() \land N \neq \text{succ}(\text{zero}())
}
\]

The first proof rule we apply is the following.

\[
\exists_R : \frac{\Gamma \vdash \phi(t/x), \Delta}{\Gamma \vdash \exists x : s, \phi, \Delta}
\]

Here we are allowed to choose a term to replace \(x\). We choose the representation for two. We then have a right hand side conjunction. The corresponding proof rule will split the proof tree.

\[
\text{\&}_R : \frac{\Gamma \vdash \phi_1, \Delta \quad \Gamma \vdash \phi_2, \Delta}{\Gamma \vdash \phi_1 \land \phi_2, \Delta}
\]

For the branch of the left conjunct, we first apply the negation proof rule

\[
\neg_R : \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg(\phi), \Delta}
\]

Then, as the terms are obviously different, the equality must be false, meaning we have a false formula on the left hand side of the sequent. We use the following axiom. (where \(op \neq op'\))

\[
\text{CINEQ} : \frac{\Gamma, \text{op}(t_1, \ldots, t_n) = \text{op}'(t'_1, \ldots, t'_n) \vdash \Delta}{\Gamma \vdash \Delta}
\]
2.3. THE VERICODE PROOF TOOL

For branch of the right conjunct we must also apply the following proof rule, otherwise the procedure is the same.

\[
\begin{align*}
\frac{\Gamma, t_1 = s_1, \ldots, t_n = s_n, \Delta}{\Gamma, \text{op}(t_1, \ldots, t_n) = \text{op}(t'_1, \ldots, t'_n), \Delta}
\end{align*}
\]

Example 2.2.13 (Outline of set membership proof)
Below follows an outline of a proof of the following semantical consequence.

\[
\models \forall S : \text{set.}\forall N : \text{nat.}\forall E : \text{nat.}((\text{Member}) N) S \implies ((\text{Member}) N) \text{pair}(E, S)
\]

For convenience, we will use the more familiar symbols \(\in\) and \(\ldots\).

\[
\frac{N \in S + N = E, N \in S}{\text{Unfolding the definition of Member}}
\]

\[
\frac{N \in S + N \in (E \cup S)}{\text{Where \(N \in S\) \implies \(N \in (E \cup S)\)}}
\]

\[
\frac{\forall \forall E : \text{nat.}N \in E \implies N \in (E \cup S)}{\forall \forall : \text{nat.}N \in S \implies N \in (E \cup S)}
\]

We start out by using the proof rule \(\forall L\) (see fig.D.6 in section D.2). This successively introduces fresh representatives for the set variable and the two natural number variables. For a discussion on fresh representatives, see fig.D.3, but roughly they represent all possible choices of terms.

Afterwards, we apply the previously presented \(\implies\) \(R\) rule.

Then we have to consider the definition of \(\in\) (Member). Some uninteresting proof steps have been omitted. Looking back at the last line of the definition in example 2.2.6 should explain the final sequent where the same formula occurs on both sides, and we can apply the \(\text{ID}\) proof rule.

2.3 The VeriCode Proof Tool

VCPT is a proof assistant for the first order \(\mu\)-calculus based on the interactive SML/NJ [SML] interpreter.

2.3.1 Constructing proofs

We construct a proof in VCPT much as described in section 2.2.3; by applying proof rules to some initial goal sequent we wish to prove valid, breaking it down to subgoals, and building a proof tree while doing so. The proof rules and axioms are called tactics in this setting and are SML-functions generally taking a sequent as argument and returning a set of subgoal sequents. For most tactics, we must supply a number for the position of the formula
to which the tactic should be applied. For instance, in the example below, we might apply
the and_r tactic on position 2.

\[- (1) 1=2, (2) 1=1 and 2=2, (3) 2=3\]

The command line interface allows us to navigate the proof tree and apply tactics to leaf
nodes. For a better overview, the proof tree can be displayed in the DaVinci graph drawing
tool [daV].

There are some subtle differences between the proof system in section D.2 and its
implementation in VCPT. For instance, when applying the exists_r (or the symmetric
forall_l) tactic, we are allowed to defer the choice of term \( t \) to substitute for \( x \). In that
case a placeholder variable is introduced for \( x \). This variable can be assigned later on in
the proof when we might have better insight regarding what term to choose. During the
assignment, the proof assistant will perform a check to ensure our choice of term would
have been legal where the variable was originally introduced. The variable might also be
assigned as a consequence of applying some tactic dealing with equalities.

Working with tactics representing only a single proof rule can be tedious. Instead we
can combine several tactics into what would correspond to derived proof rules or small
proof scripts. The combinators available are called tacticals. One highly used tactical is
t-compose_l which combines a list of tactics. When applied to a sequent, the combined
tactic gives the same subgoal sequents as applying the first tactic of the list to the current
sequent, then for each resulting subgoal sequent repeating the procedure with the rest of the
list. Another example of a tactical is t-compose, which combines a single tactic and a list
of tactics. When applied to a sequent, the combined tactic gives the same subgoal sequents
as applying the single tactic to the current sequent, and then applying the corresponding
tactic from the list to each resulting subgoal sequent. Note that intermediate sequents cre-
ated during the execution of any combined tactic, are not included in the resulting proof
tree: only the initial goal sequent, and the final subgoal sequents.

Example 2.3.1 gives a tactic formed using tacticals.

**Example 2.3.1 (Tactic for proving example 2.2.12)**

```haskell
fun proveNotZeroNorOneExample =
  t-compose_l
  [ t-compose_l [ exists_r (SOME(pt "'succ(succ(zero()))'")'),
                t-compose (and_r 1) ]
   t-compose_l [ not_r 1, eq_flat_elim_l 1 ],
   t-compose_l [ not_r 1, eq_flat_decomp_l 1, eq_flat_elim_l 1 ]
  ]
```

The sequents we have already proven valid, or which we otherwise assume to be valid,
can be stated as lemmas, as in example 2.3.2. These lemmas can then be referred to when
performing proofs and may allow us to completely validate a goal sequent, or expand the
left hand side of some goal sequent with additional implied assumptions.
2.3. THE VERICODE PROOF TOOL

Example 2.3.2 (Trivial lemma for set membership)

```
LEMMA setmembershiplemma
DECLARE N : nat, S : set IN
member N S |- forall E : nat . member N [E | S]
END
```

The mechanisms for discharge and subsumption are not tactics, since they do not operate on a single sequent. Instead they are functions operating on the constructed proof tree to determine whether the discharge condition holds. This has the consequence that we cannot combine these mechanisms using tacticals.

### 2.3.2 Terms and formulas

For giving signatures defining the relevant sorts, there is built in support for defining lists and tuples of arbitrary sorts. There is also a predefined sort for natural numbers. The somewhat cumbersome signature from example 2.2.7 for defining functions, can be rewritten as a list of pairs in VCPT as in example 2.3.3. The sets of naturals from example 2.2.2 can be written similarly simply as \textit{set} = \textit{nat list}.

Example 2.3.3 (VCPT signature for functions over naturals)

```
DATATYPES
domain = nat;
range = nat;
function = (domain*range) list
END
```

The syntax of the logic is constrained to the normal alphabet in the proof tool, but most of the formulas should be obvious. Some syntactic sugar has been added. We can write if-then-else formulas of the following shape, where \(\beta\), \(\tau\) and \(\phi\) (representing the condition, true branch, and false branch) are formulas.

```
if \(\beta\) then \(\tau\) else \(\phi\)
```

This has the same net effect as

```
(\beta => \tau) and ((not \(\beta\)) => \(\phi\))
```

Instead of the somewhat cumbersome existential quantifications and equalities used to break apart terms in the earlier examples (for instance 2.2.6), we can perform this inspection with the following construction.

```
cases S of
  [] : set => ff
| [N' | S'] => N = N' or member N S'
end
```
Finally, we can name formulas and refer to the formula by that name. This is typically how we define a relation. Below we define the relation member.

```
member : nat --> set --> prop <=
\N : nat . \S : set .
cases S of
  [ ] : set => ff
  | [N' | S'] => N = N' or member N S'
end
end
```

The <= symbol means member is a least fixed point formula. For greatest fixed point formulas we use =>, and for plain abstractions, =. Then, for instance, we can refer to the member relation when writing a goal sequent for a proof.

```
|- (1) exists N : nat. member N [1,2,3,4,5]
```

During the proof we may unfold member to consider its definition.

When naming a formula as above, we can also define local named formulas, which are only accessible to the outer formula. This is particularly useful for structuring.

```
let
  someFormula : someArgumentType --> prop <=
    ...
  where
    someInnerFormula : someArgumentType --> prop <=
      ...
end
end
```

Finally, we are also allowed to introduce a name for a relation without giving the definition as a formula. Instead we can give a more implicit definition in terms of proven or assumed lemmas. We will see examples of this in the implementation.

None of this syntactic sugar adds any new power to the logic, but it is very convenient when writing formulas and performing proofs. The various constructions are accompanied by appropriate (derived) tactics to ease their use.

### 2.3.3 Configuration

We configure VCPT for reasoning about a certain subject by writing a theory file. A theory file contains the sort signature, named formulas, references to other theory files the theory may depend on, references to files containing tactics, proven or trusted lemmas, and possibly some goal sequents we wish to prove valid.

A large part of configuring VCPT lies in writing tactics. In particular it is practical to have tactics that can prove some named formula (relation) holds between some arguments, and general heuristic tactics that take care of simple sequents.

### 2.3.4 Example proof session

Here we will perform the same proof as in example 2.2.13 using VCPT.
2.3. THE VERICODE PROOF TOOL

Our initial sequent is as follows.

\[ \vdash \{1\} \forall S : \text{set.} \forall N : \text{nat.} \forall E : \text{nat.} \quad \text{member } N \ S \implies \text{member } N \ [E | S] \]

We first apply the \texttt{forall\_l} tactic three times to introduce fresh representatives for \( S \), \( N \) and \( E \). This results in the following sequent.

\[ \vdash \{1\} \text{member } X_1 \ X_0 \implies \text{member } X_1 \ [X_2 | X_0] \]

Then, we apply the \texttt{implication\_r} tactic to give us the following.

\[ \{1\} \text{member } X_1 \ X_0 \vdash \{1\} \text{member } X_1 \ [X_2 | X_0] \]

Now we would like to reduce the right hand side formula to simply \( \text{member } X_1 \ X_0 \). We begin by unfolding the definition of \texttt{member}.

\[ \{1\} \text{member } X_1 \ X_0 \]

\[ \vdash \{1\} \text{cases } [X_2 | X_0] \text{ of } \\
\{ \[] \implies \text{ff} \\\n\{ \[N' | S'\] \implies X_1 = N' \text{ or member } X_1 \ S' \} \end{cases} \]

Here, we use the \texttt{cases\_r} tactic to handle the previously presented cases construction. As the set is not empty, we end up with the following.

\[ \{1\} \text{member } X_1 \ X_0 \vdash \{1\} X_1 = X_2 \text{ or member } X_1 \ X_0 \]

Here we apply \texttt{or\_r} to get our final sequent.

\[ \{1\} \text{member } X_1 \ X_0 \vdash \{1\} X_1 = X_2, \{2\} \text{member } X_1 \ X_0 \]

The final tactic we apply is \texttt{id}, which validates the sequent since the same formula occurs on both sides.
Chapter 3

Implementation of the $\pi$-calculus

The implementation of $\pi$-calculus in the proof assistant must cover all definitions from the background chapter. To summarise, we must find a way to represent the syntax of $\pi$-calculus processes, the transition relation, and the syntactical definitions for free names of processes, (bound) names of actions, alpha equivalence and substitution. For further motivation to our implementation decisions and presentation of alternatives and other people’s solutions, we refer to chapter 4. Our implementation of a simple specification logic is briefly presented in appendix A.

3.1 Process terms

For processes, we have chosen to follow the presentation in [SW01], reproduced in fig.2.1, and displayed earlier as a $\mu$-calculus signature in example 2.2.3. The corresponding signature in VCPT is given in fig.3.1, and a process term is given in example 3.1.1. Note that we have chosen to represent names with the built in sort of natural numbers.

```
datatype process = sum of sum
                  | compose of process * process
                  | new of name * process
                  | replication of process
                  |
      datatype sum   = stop
                  | seq of prefix * process
                  | choice of sum * sum
                  |
      datatype prefix = out of name * name
                         | inp of name * name
                         | tau
                         | match of name * name * prefix
                         |
      name          = nat
```

Figure 3.1. $\pi$-calculus process terms
Example 3.1.1 (Beverage machine from example 2.1.1)  
The simple beverage machine

\[ \text{slot(coin)}, [\text{coin } = \text{penny}] \text{ out(0)} \text{ tea} \]

can be represented as

\[ \text{sum(seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop())))))} \]

Writing process expressions as terms in the tool takes some getting used to. For convenience we developed a small parser to translate from a simpler syntax. In particular, this allows us to write processes using more easily remembered names than arbitrary numbers.

### 3.2 Syntactical definitions

For several syntactical definitions we make use of a small supplementary theory of sets of natural numbers. The set sort is, as earlier mentioned, defined as \( \text{set} = \text{nat list} \). To determine whether two set terms represent the same set, we cannot use normal term equality, for instance \([1,2]=[2,1]\) would be a false formula. Instead, set equality is given as a named formula in terms of set membership. The fact that the representation of a set is not unique has some consequences for the definitions to follow concerning the sets of names of process and action terms. For instance, we will be able to deduce that the following formula is true.

\[ \text{free_process_names} \sum\{\text{seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop()))))}\} [4,3,2,1] \]

On the other hand, the following, seemingly equivalent, formula will be false, because the \text{free_process_names} formula will consider the names of the process in a particular order.

\[ \text{free_process_names} \sum\{\text{seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop()))))}\} [1,2,3,4] \]

#### 3.2.1 Names of process terms and actions

Definitions of the sets of free names of processes and the names and bound names of actions are needed for the transition relation.

The free names of a process is defined by a named least fixed point formula. Essentially, the formula recurses through a process term, keeping track of the set of names which are currently bound, and adds any occurring name which is not currently bound, to a set of free names.

The action sort is defined as in fig.3.2. The names and bound names of actions are defined by named abstraction formulas which implement the table presented in fig.2.2.

For all these name set definitions we have created left and right hand side tactics, so we can determine the relevant set of names in a single proof step.
3.2. SYNTACTICAL DEFINITIONS

```plaintext
datatype action = fo of name * name
| bo of name * name
| bi of name * name
| silent
```

**Figure 3.2.** $\pi$-calculus action terms

```
LDMA alpha_par_1
DECLARE P:process, Q:process, P':process IN
alpha_eq P P' |- alpha_eq compose(P, Q) compose(P', Q)
END
```

**Figure 3.3.** Alpha equivalence and the composition operator

### 3.2.2 Alpha equivalence

A definition of alpha equivalence of processes is needed both for the implementation of the transition relation, and for the implementation of substitution.

Alpha equivalence is defined by a named least fixed point formula, and as a set of lemmas.

The formula definition essentially checks if two processes would have the same deBruijn representation [dB72]. In a deBruijn representation of a process, each bound name is replaced by a binding depth, which is a number counting the binders outside the binder for the name in question. All alpha equivalent processes will have the same canonical representation. The definition makes use of a small supplementary theory of functions over names, with a definition similar to the one in example 2.2.7.

The lemmas state that alpha equivalence is an equivalence relation, and capture the standard definition of alpha equivalence in terms of successive alpha conversions as in def.1. One simple lemma is given in fig.3.3.

We generally use the formula definition of alpha equivalence to determine whether two processes are alpha equivalent, and the lemmas to construct alpha equivalent processes.

Tactics have been created for easily applying the lemmas, and to determine whether two processes are alpha equivalent using the formula definition on the left and right hand side. For instance, the following sequent, representing the claim $\forall y (\forall z (\exists y 0)) =_\alpha \exists z (\forall y (\exists z y 0))$, can easily be validated.

```
|- {1} alpha_eq
    new(0,new(1,sum(seq(out(0,1),sum(stop())))))
    new(1,new(0,sum(seq(out(1,0),sum(stop())))))
```

### 3.2.3 Capture-avoiding substitution

A definition of substitution is needed for the implementation of the transition relation where we define the effect of receiving a particular name.
As briefly presented in example 2.1.12, we might have to perform an alpha conversion as part of the substitution, to avoid unintended capture.

We have defined capture-avoiding substitution \([t/z]\) in terms of non-capture-avoiding substitution, and alpha equivalence.

Our non-capture-avoiding substitution is defined by a named least fixed point formula which recurses through a process term and attempts to replace \(z\) with \(t\). If the replacement would cause an unintended binding, the replacement fails.

Our definition of capture-avoiding substitution then essentially performs as follows. It starts out by recursively replacing \(z\) with \(t\) until it finds a binder for \(t\), say \(a(t).P\). Such a binder could cause unintended capture of \(t\) if we were simply to continue replacing \(z\) with \(t\) in \(P\). To avoid such a capture, we require the subprocess \(a(t).P\) to be replaced by an alpha equivalent process, say \(a(t').P'\). To this subprocess, we then apply our non-capture-avoiding substitution to perform the rest of the substitution. It is always possible to choose \(a(t').P'\) such that this will succeed. Finally, when the substitution has been performed on the alpha equivalent subprocess, we again allow the resulting subprocess to be replaced by alpha equivalent one. This last step may allow us to reuse the name \(z\) instead of \(t'\) in the binder, if we wish.

To illustrate, we will follow this method and perform the capture avoiding substitution \([\text{coin}/\text{penny}]\) on the following process, as in example 2.1.12.

\[
\text{slot(coin),}[\text{coin} = \text{penny}]\text{spout tea.0}
\]

As the process begins with a binder for \(\text{coin}\), we begin by replacing it with an alpha equivalent process.

\[
\text{slot(foo),}[\text{foo} = \text{penny}]\text{spout tea.0}
\]

We can now perform a non-capture-avoiding substitution to replace all free occurrences of \(\text{penny}\) with \(\text{coin}\). This will result in the following process.

\[
\text{slot(foo),}[\text{foo} = \text{coin}]\text{spout tea.0}
\]

Finally, we let the result of the complete capture-avoiding substitution be any process which is alpha equivalent to the above. This could be the following, as in example 2.1.12.

\[
\text{slot(payment),}[\text{payment} = \text{coin}]\text{spout tea.0}
\]

Or, to illustrate the final point, we can reuse \(\text{penny}\) for \(\text{foo}\), allowing the following somewhat confusing example.

**Example 3.2.1 (Capture-avoiding substitution with name reuse)**

\[
(\text{slot(coin),}[\text{coin} = \text{penny}]\text{spout tea.0})[\text{coin}/\text{penny}] = \text{slot(penny),}[\text{penny} = \text{coin}]\text{spout tea.0}
\]
In any case, the result of applying our definition of substitution is a process with the free occurrences of \( z \) replaced with \( t \), and possibly with some bound names changed in the subprocesses where unintended capture could otherwise have taken place.

Tactics have been created for judging whether a substitution is correct on the left and right hand side. For instance, the following sequent can easily be validated.

\[
\{1\} \text{subst} \\
\text{sum(seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop())))))} \\
2 \\
0 \\
\text{sum(seq(inp(1,0),sum(seq(match(0,0,out(4,3)),sum(stop())))))} \\
\mid -
\]

The sequent represents the following fact from example 2.1.12.

\[
(slot(coin).\{\text{coin = penny}\}\text{spout tea}0) \neq (slot(coin).\{\text{coin = coin}\}\text{spout tea}0)
\]

### 3.3 The transition relation

The transition relation is defined by a least fixed point formula and, for efficiency reasons, a set of lemmas.

The formula definition is mostly a disjunction between the transition rules of fig.2.3, and a special case for the L-ALPHA rule of fig.2.4, which will be discussed shortly.

Roughly half of the lemmas are transcriptions of the transition rules. These lemmas are useful for deducing a particular transitions. The other half of the lemmas help us deduce all possible transitions, and in particular to prove some transition is not possible.

#### 3.3.1 L-ALPHA

As remarked in section 2.1, the L-ALPHA transition rule can be introduced as a means to perform implicit alpha conversion of processes during the deduction of a transition. This is essential for the L-CLOSE-* transition rules where we may need to alpha convert the bound names appearing in the bound output and bound input actions to ensure they agree.

For the implementation we have taken a similar approach with a slight modification.

The formula definition of the transition relation as it stands before we introduce the L-ALPHA rule or some other means of implicit alpha conversion, is roughly as illustrated below.

```ocaml
let trans : process --> action --> process --> prop <=
  \P:process . \A:action . \P':process .
  cases P of
    sum(S) => sum_trans_impl S A P'
  | compose(P1, Q1) => compose_trans_impl P1 Q1 A P'
  | new(Z, P1) => restricted_trans_impl Z P1 A P'
  | replication(P1) => replication_trans_impl P1 A P'
end
```

```ocaml
where
let compose_trans_impl :
```
CHAPTER 3. IMPLEMENTATION OF THE $\pi$-CALCULUS

\begin{verbatim}
process --> process --> action --> process --> prop =
\P:process . \Q:process . \A:action . \P':process .
\l_par_l P1 Q A P' or \l_par_r P1 Q A P' or
\l_comm_l P1 Q A P' or \l_comm_r P1 Q A P' or
\l_close_l P1 Q A P' or \l_close_r P1 Q A P'
where
\l_par_l : process --> process --> action --> process --> prop =
\P:process . \Q:process . \A:action . \P':process .
trans P1 A P1' and
bound_action_names A BAN and
free_process_names Q FN and
intersection BAN FN [] : set and
P' = compose(P1', Q)
end;
... \l_par_r, \l_comm_l, \l_comm_r, \l_close_l, \l_close_r
end;
... sum_trans_impl, restricted_trans_impl, replication_trans_impl
end

The formula inspects the shape of the process to determine what set of transition rules are applicable. As the L-ALPHA rule is applicable to any kind of process, one obvious way to introduce it would be to place it as a disjunction with the cases-construction, as follows.

let
trans : process --> action --> process --> prop <=
\P:process . \A:action . \P':process .
exists Q : process .
alpha_eq P Q and
trans Q A P
) or
cases P of
...

To simplify the automation of proofs however, we have implemented implicit alpha conversions as follows.

let
trans : process --> action --> process --> prop <=
\P:process . \A:action . \P':process .
exists Q : process .
alpha_eq P Q and
cases Q of
...

This solution will be discussed in chapter 4.

3.3.2 Tactics

For the transition relation, we argued we are mostly interested in deducing some particular transition on the right hand side, prove some transition is impossible on the left hand side, or deduce all possible transitions on the left hand side.

For the first goal we created a tactic factory which given a description of what transition rules to apply, and in what order, creates an appropriate tactic which when applied, attempts to deduce the transition.
3.3. THE TRANSITION RELATION

For instance, suppose we want to prove the following transition.

\[ \text{slot(coin).}[\text{coin = penny}] \text{spout tea}.0 \quad \text{\[\text{\rightarrow}\]} \quad [\text{penny = penny}] \text{spout tea}.0 \]
\[
\text{spout(untake)} \overset{0}{\rightarrow} \text{sink beverage}.0
\]

The corresponding sequent is as follows.

\[ |- \{1\} \text{trans} \]
\[ \text{compose(sum(seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop())))),}
\[ \text{sum(seq(out(1,2),sum(seq(inp(4,5),sum(seq(out(6,5),sum(stop()))))))))}
\]
\[ X_0^* \]
\[ X_1^* \]

Here, \(X_0^*\) and \(X_1^*\) are variables introduced instead of supplying a term when applying the \text{exists}_r tactic, as mentioned in an earlier chapter. We can now ask the factory for a tactic for applying the \text{L-COMM-R} transition rule, with \text{L-OUT} for the right hand side process and \text{L-INP} for the left hand side process. This tactic will validate the sequent and assign the variables as follows.

\[ |- \{1\} \text{trans} \]
\[ \text{compose(sum(seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop())))),}
\[ \text{sum(seq(out(1,2),sum(seq(inp(4,5),sum(seq(out(6,5),sum(stop()))))))))}
\]
\[ \text{silent()}
\]
\[ \text{compose(sum(seq(match(2,2,out(4,3)),sum(stop()))),}
\[ \text{sum(seq(inp(4,5),sum(seq(out(6,5),sum(stop()))))))}
\]

For proving some transition is impossible, we have created a tactic which applies appropriate lemmas. For instance, the following sequent can be validated since the process cannot perform any transition.

\[ \{1\} \text{trans} \]
\[ \text{new(0,new(1,sum(seq(out(0,1),sum(stop())))))}
\]
\[ X_1 \]
\[ X_0^* \]

Suppose we want to know if all possible derivatives of a process has some property. For instance, let \(\Pi\) be some particular process and \text{interesting} be some property of process terms. Our first proof goal may then be as follows.

\[ |- \{1\} \text{forall A : action . forall P' : process .}
\[ \text{trans } \Pi A P' => \text{interesting } P' \]

After some initial tactic applications we arrive at the following.

\[ \{1\} \text{trans } \Pi A P' \]
\[ |- \{1\} \text{interesting } P' \]

Here, we have created a tactic which derives all possible transitions of \(\Pi\), resulting in one new proof goal each. On these new branches, we will then attempt to prove \text{interesting} \(P'\).
Chapter 4

Rationale and alternatives

Here we motivate our implementation choices, and present alternatives and solutions from other people’s work.

4.1 The shape of process terms

There are many alternative presentations of the $\pi$-calculus in the literature. Typical variations are the shape of summands or whether summands are included at all; the shape of subprocesses in parallel composition and the replication operator; whether match prefixes eventually have to be followed by some non-match prefix; whether a mismatch prefix is included; whether action prefixes send or receive only a single or several names (that is, whether the calculus is monadic or polyadic); whether output prefixes can by followed by any process or just the inactive 0 process (asynchronous $\pi$-calculus); and how processes with possibly infinite behaviour are represented. Some presentations use the recursion construction in fig.4.1 or the equational definitions of fig.4.2 instead of the replication operator. As a further alternative, which we have chosen to avoid, we could introduce the concepts of abstractions and concretions ([Mil99]) in our syntax. In essence, when two processes communicate they first synchronise on the name used as channel. This leaves one process acting as an abstraction ready to be instantiated with some name, and one process as a dual concretion ready to supply a name. These two combine to finalise the communication.

Looking at what syntax other implementations have used, both [Rö1a, Mel94] use a more free form, presented in fig.4.3. Hirschkoff in [Hir97] skips the choice operator and the tau prefix altogether. The Mobility Workbench [Vic94] uses a polyadic variation with

\[
P ::= \ldots \\
| \text{rec } X \text{ in } P \\vspace{0.5cm}
\]

$\uparrow$ $X$ can occur in the continuation process $P$.

**Figure 4.1.** Recursion construction
concretions and abstractions and equational definitions of processes for recursion.

Given these variations in implementations and presentations there did not seem to be any particular standard to prefer. Since we can emulate the sending of multiple names [Mil99], we decided to use a monadic \( \pi \)-calculus. While polyadicity may be practical for writing down system models it would make the implementation more complex. For infinite processes, having equational or recursive definitions of processes would seem to force us to define an environment term containing definitions for all process identifiers, and then interpret the process term in relation to this environment. A replication operator is significantly simpler and, as shown in [Mil99], can be used to emulate equational definitions. All in all, the chosen presentation seemed simple enough for implementation while not being too crippled, and was already familiar to us.

### 4.2 The representation of names

The chief problems involving names in the presence of binders \((a(x).P\) and \(\forall z(P)\)) is how to represent alpha equivalence of processes and substitution. Three general approaches are presented in [Rö1b]. The shallow higher order abstract syntax approach, used for instance in [Rö1b, RHB01] works as follows. In the process syntax, the continuations \(P\) of \((a(x).P\) is a function that evaluates to an appropriately instantiated process term when applied to a name. This instantiation takes place when deriving a communication between processes, and relies on machinery provided by the proof assistant. The bound names are represented by the variables the proof assistant uses to implement these parametric terms. The benefit is that we can let the proof assistant handle substitutions and determine alpha equivalence. We cannot use this approach, as parametric terms are not available in the \(\mu\)-calculus.

Another approach which has not been considered is using deep higher order abstract syntax. For more information we refer to the above mentioned paper. The approach we have used in this project, giving concrete term representations for bound and free names and formulating predicates or lemmas that capture alpha equivalence and substitution, is called deep first order syntax.

When considering the sort of names, the primary requirement is that the set of names
4.3 Capture-avoiding substitution

Our concrete term representation of names, in particular the bound names, is the reason for our use of the alpha equivalence relation in all places where one would just implicitly change the bound names when working by hand. This is how we emulate working “up to” (see section 2.1.2) in the definition of substitution and the transition relation.

One possibly interesting alternative to our implementation would be to use deBruijn indices for all the bound names, which would make all alpha equivalent processes syntactically equivalent terms. This could also remove premises from the transition rules since the set of free and bound names are disjunct. One paper employing deBruijn indices in a deep setting is [Hir97]. We did not employ deBruijn indices because we could not determine the consequences in the outset of this project.

Some overall unsuccessful attempts were made to partition the set of names. It was believed that keeping the set of free and bound names disjunct (essentially the Barendregt variable convention from [Bar84]) could simplify the implementation of substitution, and if in addition bound names were never reused, alpha conversion could be made simpler, as well as checking the side conditions of the transition rules. We used well-formedness predicates to enforce these rules. Among other problems, the well-formedness did not behave nicely with the transition relation and alpha equivalence. We eventually dismissed the partitioning-idea altogether.

4.3 Capture-avoiding substitution

Our implementation of capture-avoiding substitution \( t/z \) is somewhat non-standard.

It was necessary to allow alpha conversions in the subprocesses where unintended capture could occur. This explains the replacement of \( a(t) \cdot P \) with the alpha equivalent \( a(t') \cdot P' \). However, we felt it unnecessary to allow alpha conversions in other unrelated parts of the process, where unintended capture could not occur, as this would only slow down certain proofs. As for the final allowed replacement with an alpha equivalent process, it was needed to allow the result in example 3.2.1.

While we have not proven so herein, we have good reason to believe this definition only allows sound substitutions. We carefully ensure no unintended capture can occur. When using the definition in the proof assistant, it must be kept in mind that changes of bound names cannot occur anywhere. For instance, the following substitution is by our definition, invalid.

\[
(\nu y (\nu z (\bar{y} t, 0))[z/t]) = \nu s (\nu z' (\bar{s} z, 0))
\]

The change of \( y \) to \( s \) during the substitution is invalid because \( y \) cannot cause unintended capture.

In any case, substitution is only used in places of the theory where this is not a problem.

For an alternative definition, a more pen-and-paper similar approach to substitution is taken in [Mel94]. There, the predicate for substitution takes a choice function as an
extra parameter. The choice function generates, given a finite set of forbidden names, some name which is not in the forbidden set. Whenever a substitution would cause an unintended capture, the choice function is employed to determine a new suitable name for the offending bound name, typically with the set of names used in the process as forbidden set. While we cannot have predicates taking functions in our logic, it may be possible to encode something similar. This solution has been overlooked.

4.4 The transition relation

The main issues to discuss is the treatment of replication and implicit alpha conversions.

4.4.1 Treating replication

Some presentations of the $\pi$-calculus give a single transition rule for replicated processes, as follows.

\[
\text{REPL}: \quad P \mid P', P' \rightarrow P''
\]

This type of transition relation definition is used in the implementations [Hir97, Mel94]. While it may seem convenient to have a single proof rule instead of three as in [SW01], there is in our opinion a drawback. Suppose we wish to prove that $P$ has no transition. We would then have to prove $P \mid P'$ has no transition. This means proving there is no $L\text{-PAR}$, $L\text{-COMM}$ or $L\text{-CLOSE}$ transition, each of involve the recurring $P'$, which brings us back to proving $P$ has no transition. To finish off the proof we must use inductive reasoning. While this reasoning may be simple, we can avoid it all together by using the three separate cases for the replication operator, in which the replication does not recur as a premise.

4.4.2 Treating implicit alpha conversions

We would like to further motivate why we avoided implementing $L\text{-ALPHA}$ as any other transition rule as follows.

\[
\text{let}
\begin{align*}
\text{trans} & : \text{process} \rightarrow \text{action} \rightarrow \text{process} \rightarrow \text{prop} \\
& \equiv \\
& \begin{cases}
\exists Q : \text{process} . \alpha = Q & \text{or} \\
\text{cases P of } & \text{cases } P \text{ of }
\end{cases}
\end{align*}
\]

Suppose we want to prove some process $\Pi$ cannot perform any transition. We would then have one branch for disproving the $L\text{-ALPHA}$ disjunct and one for the $\text{cases}$ disjunct, as illustrated in fig.4.4.

In the $\text{cases}$ branch, we would prove that no transition is possible using the current choices of bound names, and in the $L\text{-ALPHA}$ branch, we would prove that regardless of
4.4. THE TRANSITION RELATION

\[
\begin{align*}
\text{L-ALPHA} & \rightarrow \text{cases } \Pi \text{ of } \ldots \rightarrow \\
\text{L-ALPHA} & \text{ or cases } \Pi \text{ of } \ldots \rightarrow \\
& \vdash \exists A : \text{action}. \exists P' : \text{process}. \text{trans } \Pi A P' \rightarrow 
\end{align*}
\]

Figure 4.4. Branching due to L-ALPHA

how we change the bound names, still no transition is possible. As the transition relation is recursive, this branching might occur several times and become an unnecessary burden.

To be fair, the cases branch is in fact a special case of the L-ALPHA branch: the case where we choose \( Q \) equal to \( P \). This means we could start out with the L-ALPHA branch, and once it was handled, we could use the subsumption mechanism to handle the cases branch. However, as subsumption is not available as a tactic, this approach could not be easily automated. An alternative is to avoid subsumption and perform the cases proof explicitly, which would mean some duplicated work.

These small issues are avoided by the solution we used, reproduced below, and gives equivalent results.

\[
\text{let trans : process } \rightarrow \text{ action } \rightarrow \text{ process } \rightarrow \text{ prop } <= \\
\forall P : \text{ process }. \forall A : \text{ action }. \forall P' : \text{ process } . \\
\exists Q : \text{ process } . \\
\alpha eq P Q \land \\
\text{cases } Q \text{ of } \\
\ldots 
\]

Now, when proving \( \Pi \) cannot perform any transition, we can infer that \( Q \) must have the same overall shape as \( \Pi \) but with fresh representatives in place for the bound names. Effectively, \( Q \) will represent all alpha equivalent variations of \( \Pi \). We then prove this \( Q \) cannot perform any transition. During this proof, there are no awkward L-ALPHA branches, but possibly some extra proof goals regarding which names are free and bound. In any case, these are handled by the tactics.
Chapter 5

Proofs using the implementation

In the following sections, we present a few proofs making use of our implementation. We will begin with a very simple specification to demonstrate substitution and the transition relation. Then we prove a simple transition relation lemma, which we will make use of in the final example with a simple specification of infinite behaviour. This last example illustrates compositional reasoning in VCPT.

5.1 A simple specification

Suppose we wish to prove we can insert something through the slot of our beverage machine, which causes it to stop completely. The \( \pi \)-calculus process, and its representation in the implementation, is reproduced below.

\[
\text{slot(coin).} \quad \text{coin = penny} \quad \text{spout tea.0} \\
\text{sum(seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop())))))}
\]

Translating the specification to our \( \pi \)-calculus implementation, we wish to prove the beverage machine can perform an input transition, and for some instantiation of \( \text{coin} \), the resulting process cannot perform any further transitions. Our first proof goal is as follows.

\[
|\text{-} (1) \text{exists Payment:name. exists P':process.} \\
\text{trans} \\
\text{sum(seq(inp(1,0),sum(seq(match(0,2,out(4,3)),sum(stop())))))} \\
\text{bi(1,0)} \\
p' \\
\text{and} \\
\text{forall P'':process.} \\
\text{subst P' 0 Payment P'' =>} \\
\text{not (exists A:action. exists P''':process. trans P'' A P''')} 
\]

The overall proof strategy is to choose something different than 2 (representing \textit{penny}) for \textit{Payment}, and then simply perform the substitution and transitions.

We begin by choosing the number 4711 for \textit{Payment}. We then introduce a variable for the resulting process \( P' \) of the initial input transition. After splitting the conjunction, the proof goal for input transition is as follows.
Here we simply apply a transition tactic to deduce the only possible transition and validate the sequent.

The next proof goal corresponds to the other conjunct. Note that $X_0$, the variable that was introduced for $P'$ above, was instantiated according to the first transition.

We begin by introducing a fresh representative for $P''$, the result of applying the substitution. We will then prove the resulting process of this substitution cannot perform any further transitions.

Here we are to prove the process cannot perform any further transitions. This goal is simply validated by the tactic for disproving transitions, which concludes the proof.

### 5.2 A transition relation lemma

We will prove the following simple lemma.

$$ P \xrightarrow{a} P' \Rightarrow 0 \parallel P \xrightarrow{a} 0 \parallel P' $$

Our initial sequent corresponding to this lemma is as follows.

The strategy is to consider all possible actions the process represented by $P$ can perform, and show that these actions can be mirrored by letting the composition process perform the same action using the $L$-PAR-R transition rule.
5.2. A TRANSITION RELATION LEMMA

To consider the actions, we must add a formula with type information for \( A \). A type information formula describes the structure of a term according to its sort.

After unfolding this type information formula and applying the argument \( A \), we have the following sequent.

\[
\begin{align*}
\{1\} & \text{trans } P \ A \ P', \\
\{2\} & \exists X_2: \text{name} . \exists X_3: \text{name}. \\
& \quad \text{A=fo}(X_2,X_3) \quad \text{and} \\
& \quad \{\text{lfp } X_7: \text{nat} \rightarrow \text{prop.<...>}\} \ X_2 \quad \text{and} \\
& \quad \{\text{lfp } X_{11}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_3 \\
\text{or} \\
\{\text{lfp } X_7: \text{nat} \rightarrow \text{prop.<...>}\} \ X_2 \quad \text{and} \\
& \quad \{\text{lfp } X_{11}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_3 \\
\text{or} \\
\{\text{lfp } X_{17}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_2 \quad \text{and} \\
& \quad \{\text{lfp } X_{21}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_3 \\
\text{or} \\
& \quad \text{A=silent()} \\
\vdash \{1\} \text{trans} \ \text{compose}(\text{sum}(\text{stop}()),P) \ A \ \text{compose}(\text{sum}(\text{stop}()),P')
\end{align*}
\]

The type information requires \( A \) to be one of the four actions, as defined in fig 3.2. The least fixed point formulas, beginning with \( \text{lfp} \), simply requires the names in the actions to be natural numbers. The next step will now be to split the disjunctions, which will create one branch for each possible type of action. As these branches are very similar, differing only in the type of action, we only consider the first branch where the action is free output.

\[
\begin{align*}
\{1\} & \text{trans } P \ A \ P', \\
\{2\} & \exists X_2: \text{name} . \exists X_3: \text{name}. \\
& \quad \text{A=fo}(X_32,X_33) \quad \text{and} \\
& \quad \{\text{lfp } X_{16}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_32 \\
& \quad \{\text{lfp } X_{20}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_33 \\
\vdash \{1\} \text{trans} \ \text{compose}(\text{sum}(\text{stop}()),P) \ \text{fo}(X_32,X_33) \ \text{compose}(\text{sum}(\text{stop}()),P')
\end{align*}
\]

Here we introduce two fresh representatives for the possible names used in the free output action, and then instantiate \( A \) accordingly.

\[
\begin{align*}
\{1\} & \text{trans } P \ \text{fo}(X_{32},X_{33}) \ P', \\
\{2\} & \{\text{lfp } X_{16}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_{32} \\
& \quad \{\text{lfp } X_{20}: \text{nat} \rightarrow \text{prop.<...>}\} \ X_{33} \\
\vdash \{1\} \text{trans} \ \text{compose}(\text{sum}(\text{stop}()),P) \ \text{fo}(X_{32},X_{33}) \ \text{compose}(\text{sum}(\text{stop}()),P')
\end{align*}
\]

Now, we have the premise that \( P \) can perform a free output and become \( P' \). Furthermore, the left hand side process of the composition does not have any free names, so all conditions for applying the L-PAR-R transition rule are satisfied.
Our tactic factory cannot completely handle the deduction of this transition, but it can get us as far as the following.

\[
\begin{align*}
1 & \text{trans } P \text{ fo}(X_{32},X_{33}) P', \\
2 & \text{(lfp } X_7 : \text{nat }\rightarrow \text{prop.<...>) } X_{32} \text{ and} \\
& \text{ (lfp } X_11 : \text{nat }\rightarrow \text{prop.<...>) } X_{33} \\
\vdash & \text{ trans compose(sum(stop()),P) fo}(X_{32},X_{33}) \text{ compose(sum(stop()),P')}, \\
2 & \text{trans } P \text{ fo}(X_{32},X_{33}) P'
\end{align*}
\]

Here, the same formula occurs on both sides of the turnstile, and we are finished.

### 5.3 Infinite shout

We will prove that the process $!x.y.0$ satisfies the following specification.\(^1\)

\[
\begin{align*}
\nu \text{shout} : \text{process }&\rightarrow \text{prop}. \\
\lambda P : \text{process}. \\
\exists P' : \text{process}. \\
((\text{trans } P \bar{x} y) P' \land (\text{shout} P')
\end{align*}
\]

The specification means the process can perform an infinite number of free outputs of $y$ along $x$. This proof is a bit more complicated than earlier proofs, because it involves fixed points and the concept of compositional reasoning with the TERM CUTF proof rule. The overall proof strategy is as follows.

First we prove that the process can perform the free output transition. This will result in a derivative process $0 !x.y.0$. The goal is then to prove that this process in turn satisfies the specification, which entails proving $0 | \ldots | 0 !x.y.0$ satisfies the specification, and so on. We prove this by a form of induction for “if $X : \text{shout}$ then $0 | X : \text{shout}$”.

For simplicity, we have given the specification formula a name, $\text{shout}$ in VCPT.

\[
\begin{align*}
\text{shout} : \text{process }&\rightarrow \text{prop} =>\\
\forall P : \text{process} . \\
\exists P' : \text{process} . \\
\text{trans } P \text{ fo}(0,1) P' \text{ and shout } P'
\end{align*}
\]

With the sugared syntax $t : \phi$ for applications ($\phi$) $t$, our initial proof goal is as follows.

\[
\vdash (1) \text{ replication(sum(seq(out(0,1),sum(stop())))}) : \text{shout}
\]

Our first step will be to introduce an approximation ordinal for the greatest fixed point formula $\text{shout}$, using a tactic corresponding to the proof rule APPROX\(_R\). If

\[
\vdash (1) \text{ replication(sum(seq(out(0,1),sum(stop())))}) : \text{shout(X_0)} \uparrow
\]

We then unfold the definition of $\text{shout}$, using a tactic for \(\nu\text{UNFOLD}2_R\) and apply the

\[^1\text{The free output action } \bar{x} y \text{ is for instance represented by } \text{freeout}(x, y)\]

44
5.3. INFINITE SHOUT

process term argument.

\[
\begin{align*}
\{1\} & \quad \text{exists } P': \text{process.} \\
& \quad \text{trans } \text{replication(sum(seq(out(0,1),sum(stop()))))) fo(0,1) P'} \\
& \quad \text{and shout}(X_1) P'
\end{align*}
\]

Here we introduce a variable for \( P' \) and split the conjunction. In the proof goal for the transition relation, we easily derive the only possible transition. The \text{shout} goal is as follows.

\[
\begin{align*}
\{1\} & \quad \text{shout}(X_1) \text{ compose}(\text{sum(stop()),x}) \\
& \quad \text{replication(sum(seq(out(0,1),sum(stop())))))}
\end{align*}
\]

Comparing this goal the previous goal marked \( \dagger \), we see that the approximation ordinal has been decreased, and that the new process term now includes an inactive 0.² Informally, we suspect that if \( \exists y.0 \) satisfies the specification, then surely 0 \( !\exists y.0 \) does as well, since 0 cannot interfere in any way. We can formalise this reasoning by applying tactics for the \text{TERMCUT} proof rule, reproduced below.

\[
\text{TERMCUT:} \quad \begin{array}{c}
\text{split } \Gamma \vdash t : \phi, \Delta \\
\Gamma, x : \psi \vdash t/x : \phi, \Delta \\
\Gamma \vdash t[t'/x] : \phi, \Delta \\
x \text{ fresh}
\end{array}
\]

Essentially, the \text{TERMCUT} proof rule allows us to prove a property of some term by using a property of a subterm as a leverage. Here, we will use \( !\exists y.0 : \text{shout}(X_1) \) to show 0 \( !\exists y.0 : \text{shout}(X_1) \). This corresponds to choosing \text{shout for } \phi \text{ and the following term for } t'.

\[
\begin{align*}
\text{replication(sum(seq(out(0,1),sum(stop())))))}
\end{align*}
\]

The expression \( t[t'/x] \) would correspond to the complete process term. This will result in two new goal sequents, one of which is as follows.

\[
\begin{align*}
\{1\} & \quad \text{replication(sum(seq(out(0,1),sum(stop())))))} : \text{shout}(X_1)
\end{align*}
\]

Here we are to prove that \( !\exists y.0 : \text{shout}(X_1) \). This sequent is clearly a new occurrence of the earlier sequent \( \dagger \), and as \( X_1 < X_0 \), we can discharge this goal.

The other goal sequent begins a more complicated branch where we perform the previously mentioned induction.

\[
\begin{align*}
\{1\} & \quad \text{shout}(X_1) \text{ compose}(\text{sum(stop())),X_{51})
\end{align*}
\]

Here, we are to prove 0 \( X : \text{shout}(X_1) \), given the leverage \( X : \text{shout}(X_1) \).

²The sugared notation has disappeared as well
The overall goal is to transform this sequent to a similar sequent with another representative in place for $X_{51}$ and a decreased approximation ordinal in place for $X_{1}$, in order to use the discharge mechanism again.

The first step is to unfold the right hand side $\text{shout}$ formula and apply the process term argument.

$$
\begin{align*}
&\{1\} \ X_{1} < X_{0}, \ 2 \ X_{51} : \text{shout}(X_{1}), \ 3 \ X_{52} < X_{1} \\
&\mid - \\
&\{1\} \ \exists P' : \text{process}. \\
&\quad \text{trans compose}(\text{sum}(\text{stop}()), X_{51}) \ fo(0,1) \ P' \\
&\quad \text{and shout}(X_{52}) \ P' \\
\end{align*}
$$

As seen, this introduces the new approximation ordinal $X_{52}$, less than the previous $X_{1}$. We would like a $\text{shout}$ formula with this approximation ordinal on the left hand side as well.

To achieve this, we unfold the left hand side $\text{shout}$ formula. When doing so (using a tactic corresponding to proof rule $\nu\text{UNFOLD}3_L$), we have the option of specifying what approximation to use in the resulting formula - provided we can prove it is less than the current $X_{1}$. We choose $X_{52}$.

The unfolding produces two new goal sequents. In the first, we get $X_{52} < X_{1}$ on the right hand side, and as it already occurs on the left hand side, the sequent is trivially valid.

After applying the arguments to the $\text{shout}$ formulas, the other goal sequent is as follows.

$$
\begin{align*}
&\{1\} \ X_{1} < X_{0}, \\
&\{2\} \ \exists P' : \text{process}. \\
&\quad \text{trans} X_{51} \ fo(0,1) \ P' \\
&\quad \text{and shout}(X_{52}) \ P', \\
&\{3\} \ X_{52} < X_{1} \\
&\mid - \\
&\{1\} \ \exists P' : \text{process}. \\
&\quad \text{trans compose}(\text{sum}(\text{stop}()), X_{51}) \ fo(0,1) \ P' \\
&\quad \text{and shout}(X_{52}) \ P' \\
\end{align*}
$$

Now, by introducing a fresh representative $X_{53}$ for $P'$ on the left hand side, choosing the term $\text{compose}(\text{sum}(\text{stop}()), X_{53})$ for $P'$ on the right hand side, and finally splitting the conjunctions on the left and right hand side, we get two final sequents.

The first sequent is as follows.

$$
\begin{align*}
&\{1\} \ X_{1} < X_{0}, \\
&\{2\} \ \text{trans} X_{51} \\
&\quad \text{fo}(0,1) \\
&\quad X_{53}, \\
&\{3\} \ \text{shout}(X_{52}) \ X_{53}, \\
&\{4\} \ X_{52} < X_{1} \\
&\mid - \\
&\{1\} \ \text{trans} \\
&\quad \text{compose}(\text{sum}(\text{stop}()), X_{51}) \\
&\quad \text{fo}(0,1) \\
&\quad \text{compose}(\text{sum}(\text{stop}()), X_{53}) \\
\end{align*}
$$

Looking at the two $\text{trans}$ formulas, we see that they are in fact an instance of a the previously proven lemma, meaning this sequent is valid.
5.3. INFINITE SHOUT

The other sequent is as follows.

\[
\begin{align*}
(1) \ & X_1 < X_0, \\
(2) \ & \text{trans} \\
& \quad X_{51} \\
& \quad f_0(0,1) \\
& \quad X_{53}, \\
(3) \ & \text{shout}(X_{52}) \ X_{53}, \\
(4) \ & X_{52} < X_1 \\
| & \\
(1) \ & \text{shout}(X_{52}) \ \text{compose}(\text{sum}(\text{stop}()),X_{53})
\end{align*}
\]

This sequent is essentially the same as the sequent marked by $\S$, with approximation ordinals decreased as required. This allows us to use the discharge mechanism, which concludes the proof.
Chapter 6

Conclusion and future work

Returning to the motivating problem of this thesis, whether we can use VCPT for reasoning about behavioural specifications of $\pi$-calculus processes, it has been demonstrated that we can, at least for simple cases.

A representation of $\pi$-calculus processes has been given as a sort in the logic, with natural numbers for names. The basic syntactical operations concerning names, substitution and alpha equivalence have been implemented as suitable abstraction or fixed point formulas. Lemmas have also been given for alpha equivalence and alpha conversion. The transition relation has been implemented as a rather large least fixed point formula, and as a set of lemmas, mainly for efficiency reasons. We allow alpha conversions during the derivation of a transition in a required initial step, as opposed to introducing an L-ALPHA transition rule.

Tactics have been created to easily handle proof goals concerning the above operations. In particular, we have created tactics for the transition relation. First, a tactic for deriving a specific transition, specified by the user as the names of the transition rules to apply. Second, a complementary tactic for deriving all possible transitions, or to prove the impossibility of a transition.

Overall we claim this allows reasoning on a fairly comfortable level, and that the main difficulty of using this implementation is learning how to use the proof system and the facilities of the proof assistant.

In the following sections we suggest future work with the implementation and the proof assistant.

6.1 Future work with the implementation

While we have not conducted extensive experiments to support the claim, it becomes apparent when using the implementation that it does not perform very well for more realistically sized problems. There are several things that could be done to improve performance, and we will suggest a few.

Other possible tasks are to formalise further $\pi$-calculus concepts, or to investigate other approaches to implementing the theory in order to avoid some of the problems we have had
to handle.

6.1.1 Improving performance

**Lemmas** Performance problems were first noted when using the formula definition of the transition relation to prove or disprove transitions. The main cause was essentially the combination of lengthy formulas, and the introduction of and assignment to, variables and fresh representatives. This was the motivating reason for giving a secondary definition of the transition relation as a set of lemmas, which turned out to give a dramatic improvement. For instance, an earlier example where we proved $\forall z (\exists y(z \equiv 0))$ cannot perform any transition took roughly 200 seconds using tactics for the formula definition, and 3 seconds with tactics for applying the proper lemmas. It should be noted that the transition relation lemmas have all been proven to agree with the formula definition.

We believe one worthwhile goal would be to provide lemmas for other defined relations, and smart tactics to apply the proper ones, instead of working through the definitions.

For larger proofs, an extensive suite of proven lemmas for typical situations would be very useful to significantly shorten proofs. In section 5.2 we presented one very simple example.

**Deep tactics** Some defined relations are simple to prove or disprove informally, but require several proof steps in the proof assistant. While these proof steps are hidden by tactics, they are time consuming. This is apparent in the two simple but often occurring cases of proving some element is a member of a set, or proving some element is not a member of a set. Our standard tactics for determining set membership works through the definition (presented in section 2.3.2), which potentially creates a temporary proof tree as deep as the number of members in the set. We can achieve the same effect in a faster, somewhat less formal way. The terms (processes, actions, sets) in our implementation are fundamentally of some SML type. This means we can extract the set and the questioned member from the set membership formula, and determine the set membership in straightforward SML. Of course, care must be taken to achieve the same net result as the normal tactics in order not to break the soundness of the proof system, i.e. the same subgoal sequents must be created, if any.

We have provided these faster alternative tactics for set membership. One potential performance improvement could be to handle other simple relations the same way.

**Readability vs. Effectiveness** When creating this implementation, we preferred being somewhat self-documenting in our definitions rather than writing definitions that were effective from a proof perspective. For instance, we used several local named formulas to structure definitions, which requires more unfoldings and instantiations of variables. Also, consider following excerpt from the implementation of the transition rule L-PAR-L.

```plaintext
... bound_action_names A BAN and
```

1 2.2 GHz AMD Athlon 64 3500+ 1 GB
6.2. FUTURE WORK WITH THE PROOF ASSISTANT

Here, instead of extracting a set term from the process $Q$ and a set term from the action $A$ and performing a set intersection test, we could have written a formula that directly inspects the process term to see if some name occurs freely. Similar solutions could of course be adopted in other places where we preferred being more explicit about what is going on.

6.1.2 Extending the theory

First we could make better use of the current definitions by providing more lemmas. Not only for possible performance improvements, but because they are more convenient to work with.

Second, we could introduce new definitions. When studying the behaviour of processes, it is not uncommon to work up to structural congruence or up to some form of bisimulation, so these notions would be interesting to introduce. As the transition relation definition stands now, we can fairly automatically decide whether some particular transition is possible. Whether this is still possible if we introduce structural congruence, as a transition rule or otherwise, would be an interesting and probably difficult problem.

6.1.3 Re-implementing

Many of our performance problems and the awkwardness of some definitions (read: substitution) are the result of our explicit treatment of bound names and alpha equivalence. An implementation using for instance deBruijn indices might avoid these problems. Some syntactical definitions for updating the indices will likely be necessary, but overall the theory might become simpler.

Overall, it would be interesting to see what problems can be avoided, and what new ones emerge if we implement the $\pi$-calculus using other approaches.

6.2 Future work with the proof assistant

When it comes to the proof assistant, it seems clear that it is not currently in wide use or active development, so there has been no real drive to provide practical usage support. However, some suggestions are presented below.

6.2.1 Documentation

While it may not be the most exiting task, an up-to-date user’s manual and thorough tutorial on the proof assistant would be a great contribution if it is to be of public use. As it stands now, the workings of the proof assistant seems to be passed on from user to user, and often, the answer to your question is in the source code.
6.2.2 A tool for tactic drafting

As we mentioned in section 2.3.3, a large part of configuring VCPT lies in writing tactics. Having written quite a few (counting failed attempts), we are convinced that at least the creation of tactics for simple definitions could be automated, and that a lot of boilerplate could be generated for more complex definitions. This is certainly no small task, but we believe it would be of great use.

6.2.3 Review of substitution code

The proof assistant does not really cope well with lengthy formulas when a large number of assignments to variables and fresh representatives take place. When an assignment, say $x = t$, takes place, the proof assistant performs a substitution $\{t/x\}$ on all the formulas occurring in the sequent. If these formulas are large, it naturally takes some time to traverse their tree-shaped representation.

This is a good reason to keep definitions short or use lemmas, but we are not certain this will always work. For instance, suppose we have a named lengthy least fixed point formula defining some relation (say, $\text{trans}$), and we want to make use of the fact that we are only allowed to recurse through the formula a finite number of times. For this we would use approximation ordinals. Now, when giving the relation as a set of lemmas, it does not seem to be possible to refer to ordinals at all, making this kind of proof impossible to perform without the formula definition.

In any case, we believe it might be interesting to reconsider the way variables and fresh representatives are represented and assigned in the proof assistant.

6.2.4 Correcting possible bugs

In appendix F we present a bug to be fixed. Bugs in proof assistants are serious since they may render earlier performed proofs suspect, so a second look through the source code would be worthwhile. Perhaps the most effective way to wash out further bugs is to make the tool and source code publicly available.
Bibliography


[FGN+] Lars-Åke Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for Erlang. citeeseer.ist.psu.edu/fredlund03verification.html.


53


Appendix A

MPW-logic

Here we present a modal logic suitable for simple behavioural specifications of $\pi$-calculus processes, followed by our implementation of the logic in the proof assistant.

A.1 Overview

The logic, whose syntax is given in fig.A.1, is a variation of a logic presented in [MPW93]. We refer to it as MPW-logic. The primary difference is that in this presentation, conjunctions apply only to two formulas at a time and we only allow finite formulas, whereas the original presentation allowed conjunctions of a possibly infinite set of formulas. Our simplification makes the logic significantly less powerful. In particular, a specification formula can only refer to a finite part of the execution of a process. The formulas which mention the bound input or bound output action, bind the parenthesized name with scope $\varphi$. The notions of substitution and free and bound names are then defined as expected. The bound names all act as placeholders. We are allowed to perform implicit alpha conversions of both formulas and processes at any time.

If a process $P$ satisfies a specification formula $\varphi$, we write $P \models \varphi$, where $\models$ is the so called satisfaction relation. This relation is defined in fig.A.2.

The need for the various bound input constructions are well motivated in [MPW93]. A

\[
\varphi ::= \text{true} \\
\quad \varphi_1 \land \varphi_2 \\
\quad \neg \varphi \\
\quad [a = b] \varphi \\
\quad \langle \alpha \rangle \varphi \quad \dagger \\
\quad \langle \alpha(y) \rangle^E \varphi \\
\quad \langle x(y) \rangle^E \varphi
\]

$\dagger$ $\alpha$ is an action from the $\pi$-calculus.

Figure A.1. Syntax of the MPW-logic
few proofs are reproduced below.

**Example A.1.1 (Plain input)**

\[ x(y), [x = y] \tau.0 \models \langle x(y) \rangle \neg((\tau) \text{true}) \]

*The specification says the process can perform an input along x after which no silent transition is possible. We choose to input anything but x, say t, after which we must prove the following.*

\[ [x = t] \tau.0 \models \neg((\tau) \text{true}) \]

*Meaning we must disprove the following.*

\[ [x = t] \tau.0 \models \langle \tau \rangle \text{true} \]

*This is easily done, because the process can not perform any action as a result of the match prefix.*

**Example A.1.2 (Early input)**

\[ x(y), [y = u] \tau.0 + x(y), [y = v] \tau.0 \models \langle x(y) \rangle E \neg((\tau) \text{true}) \]

*The specification says that for each name, the process can perform some input action to receive the name along x, after which no silent transition is possible.*

*The key to the proof is to consider what names can be input and then for each possible name choosing an appropriate transition. When receiving the name u the process must behave as the right hand side of the choice construction \( (x(y), [y = v] \tau.0), \) and when receiving*
A.2. IMPLEMENTATION

the name \( v \), the process must behave as the left hand side. Suppose we are proving the specification is satisfied for the case where the name \( u \) is received. We then perform the right hand side input, after which we must prove the following.

\[ [u = v].0 \models \neg((\tau)\text{true}) \]

Which we handle just as in the previous example.

Example A.1.3 (Late input)

\[ x(y).0 \models \langle x(y) \rangle^{\perp} - ((\tau)\text{true}) \]

The specification says that the process can perform some input transition to receive any name, after which no silent transition is possible.

In this case, only one input transition is possible for the process, and afterwards, no transition at all is possible, so the result follows shortly after we have reached the following formula.

\[ 0 \models \neg((\tau)\text{true}) \]

A.2 Implementation

We provide a translation from MPW-logic formulas to \( \mu \)-calculus formulas. The main problems of the translation are the substitutions for instantiating placeholder names with names received or bound names sent, and the use of a symbol \( fn \) to refer to the free names of a subformula.

To achieve the same result as substitution over formulas, whenever some name is received or some bound name is sent, we successively build a function term (signature according to section 2.3.2, evaluation defined similar to example 2.2.7) which relates the placeholder name and the replacement. Every placeholder name in the continuation of the formula is then interpreted with respect to this function term.

The symbol \( fn \) is used to avoid choosing a \( w \) which occurs freely, with exception for \( y \), in the ensuing subformula when handling a bound output. This is to ensure the name being output is truly new.

The free names of the subformula are names which have always been free, and names which are placeholders. In the translation of a bound output formula, we explicitly embed a set term with the free non-placeholder names of the subformula, and a set term with the free placeholder names of the subformula. We then ensure that \( w \) is different from all the non-placeholder names, and also that none of the placeholder names would be evaluated to \( w \) according to the earlier mentioned function term.

The complete translation is given in fig.A.3. The symbol \( \beta \) is the set of names bound at the current point in the translation. We read \( \Gamma(A,\beta)^{\tau} \) as the translation of \( A \) when considering the names in \( \beta \) bound.

To prove \( P \models A \), we attempt to prove the following \( \mu \)-calculus formula.

\[ ((\Gamma(A,\emptyset)^{\tau}) \ P) [] \]
The notation $x_Q$ should be read as $x$ when $x \notin \beta$, otherwise we should have an evaluation step $\exists N : name.(((eval) g) x) N$ before, and use $N$ for $x_Q$. The set $\Phi$ is the set of free names which are not placeholders occurring in the subformula $A$. The set $\Lambda$ is the set of free placeholders occurring in the subformula, excluding the placeholder for the bound output connective.

**Figure A.3.** MPW to $\mu$-calculus translation
Appendix B

Fixed point theory

This material is condensed from [Win93, Phi92, Kre03] and is not intended to be the only source of information for a novice to the subject, but a short summary to make this thesis more self-contained.

A fixed point of a function \( f : D \rightarrow D \) is an element \( x \in D \) such that \( f(x) = x \). There may be multiple fixed points for a given function. Typically we are interested in the least- and greatest fixed point, which we can define given a suitable ordering \( \subseteq \). For a least fixed point \( x \), \( x \subseteq y \) for all other fixed points \( y \), and for a greatest fixed point we have \( y \subseteq x \).

In this thesis we are mainly interested in the fixed points of functions with functions as domain and range, \( f : [X \rightarrow Y] \rightarrow [X \rightarrow Y] \). This will turn up in the presentation of the semantics of first order \( \mu \)-calculus.

For stating necessary conditions for the existence of fixed points we will need the following definitions.

**Definition 2 (Partial order)** A partial order \( (D, \sqsubseteq) \) is a set \( D \) with a binary relation \( \sqsubseteq \) which is reflexive, transitive and antisymmetric:

- \( \forall x \in D . x \sqsubseteq x \)
- \( \forall x, y, z \in D . x \sqsubseteq y \land y \sqsubseteq z \implies x \sqsubseteq z \)
- \( \forall x, y \in D . x \sqsubseteq y \land y \sqsubseteq x \implies x = y \)

**Example B.1 (Partial order)** The natural numbers under the usual less-than-or-equal relation, \( (\mathbb{N}, \leq) \) is a partial order. \( (\mathbb{N}, \geq) \) as well.

**Definition 3 (Bounds)** Given a partial order \( (D, \sqsubseteq) \) and a subset \( X \subseteq D \), \( p \in D \) is an upper bound of \( X \) if \( \forall q \in X . q \sqsubseteq p \), and a lower bound if \( \forall q \in X . p \sqsubseteq q \). The most interesting bounds are the least upper bound and the greatest lower bound.

A least upper bound \( u \) of \( X \), written \( \sqcup X \), is an upper bound such that for all upper bounds \( p \) of \( X \), \( u \subseteq p \). For a greatest lower bound \( l \), \( \sqcap X \), we have \( p \subseteq l \) for all lower bounds \( p \). If the least upper (greatest lower) bound exists, it is unique.
**APPENDIX B. FIXED POINT THEORY**

**Definition 4 (Complete partial order)** Given a partial order \((D, \subseteq)\), a subset \(C \subseteq D\) is a chain if its elements can be linearly ordered, that is, for every \(x, y \in C\) we have \(x \subseteq y\) or \(y \subseteq x\). An \(\omega\)-chain is an infinite chain:

\[
d_1 \subseteq d_2 \ldots \subseteq d_n \ldots
\]

An element is called a bottom and is typically written \(\bot\) if \(\forall x \in D. \bot \subseteq x\). If a bottom exists, it is unique. Similarly, we can define a top, \(\top\).

If a partial order has a bottom element and all \(\omega\)-chains each have a least upper bound, the partial order is complete (or complete with bottom).

**Example B.2 (Complete partial order)** Returning to previous examples, \((\mathbb{N}, \leq)\) has a bottom element, 0, but no upper bound for the infinite chain:

\[
0 \leq 1 \leq 2 \ldots
\]

If we form \((\mathbb{N} \cup \{\infty\}, \leq)\) and extend the meaning of \(\leq\) to mean \(n \leq \infty\), this would be a complete partial order with bottom.

Looking at \((\mathbb{N}, \geq)\), this partial order has a “greatest” element 0 guaranteeing there is a least upper bound for each chain.

\[
n \geq n - 1 \geq \ldots 0
\]

This qualifies for being a complete partial order, but bottomless.

**Definition 5 (Lattice)** A lattice is a partial order \((L, \sqsubseteq)\) for which each finite subset \(C \subseteq L\) has a least upper bound and a greatest lower bound. If this holds also for infinite subsets, the lattice is complete.

**Example B.3 (Powerset lattice)** For any set \(S\), \((\mathcal{P}(S), \subseteq)\) forms a complete lattice with bottom element \(\emptyset\). The least upper bound for \(C \subseteq \mathcal{P}(S)\) is given by \(\bigcup C\), and the greatest lower bound by \(\bigcap C\). \(S\) itself is the top element.

**Definition 6 (Monotonicity)** Given two complete partial orders \((D, \subseteq_D)\) and \((E, \subseteq_E)\) and a function \(f : D \to E\), we say \(f\) is monotonic, or order preserving, iff

\[
\forall d_1, d_2 \in D. d_1 \sqsubseteq_D d_2 \implies f(d_1) \sqsubseteq_E f(d_2)
\]

**Definition 7 (Pre- and postfixed points)** Say \(x\) is a prefixed point if \(f(x) \subseteq x\), and a postfixed point if \(x \subseteq f(x)\).

Now we can state necessary conditions for the existence of fixed points as follows.

**Definition 8 (Knaster-Tarski fixed point theorem)** If \((L, \sqsubseteq_L)\) is a complete lattice and \(f : L \to L\) a monotonic function, then the least fixed point of \(f\) can be defined as:

\[
\mu x := \bigcap \{x \in L \mid f(x) \sqsubseteq_L x\}
\]
Similarly, for the greatest fixed point:

$$\forall x \in L \{ x \in L \mid x \subseteq_L f(x) \}$$

Alternate more constructive definitions in terms of transfinite ordinals can also be given. For finite domains, we can use these definitions to find the least (greatest) fixed points.

**Definition 9 (Alternate fixed point definition (also Knaster-Tarski))** Given a complete lattice and function as above, the infinite increasing sequence $x_{\alpha \in \text{Ord}}$ in fig. B.1 will reach the least fixed point of $f$.

- $x_0 = \bot$
- $x_{\alpha+1} = f(x_{\alpha})$
- $x_{\lambda} = \bigcup_{\alpha < \lambda} x_{\alpha}$
  ($\lambda$ is a limit ordinal)

**Figure B.1.** Sequence with least fixed point as limit

Dually we find the greatest fixed point as the limit of the sequence in fig. B.2

- $x_0 = \top$
- $x_{\alpha+1} = f(x_{\alpha})$
- $x_{\lambda} = \bigcap_{\alpha < \lambda} x_{\alpha}$

**Figure B.2.** Sequence with greatest fixed point as limit
Appendix C

Many-sorted first order logic

This section presents many-sorted first order logic, and is condensed from [RS92, Pel01, Kai04]. Similarly to the fixed point section, we do not recommend this section to be the only source of information for the novice. The purpose of the section is primarily to pave way for the μ-calculus presentation.

First order logic allows us to reason about relations and individuals. We first present its syntax, followed by a semantics and in the notion of semantic consequence.

C.1 Syntax

The syntax of first order logic is flexible in that it contains of a fixed logical part containing the usual connectives and quantifiers, and an application specific non-logical part which defines relations, sorts and representatives for the various individuals we reason about.

This non-logical part is defined by a vocabulary (or signature), given as a tuple shown below.

\[(F, R, a, S, s)\]

Here, \(F\) is a set of function symbols (\(0\), \(successor\), \(top\), \(pop\)), \(R\) is a set of relation symbols (\(\leq\), \(instanceof\)), \(a\) is a function that assigns an arity (a natural number) to each of the function- and relation symbols, \(S\) is a set of sorts (\(int\), \(nat\), \(stack\)), and \(s\) is a function that assigns a sort from \(S\) to each argument of a function or relation symbol, and additionally to the result sort of each function symbol. Function symbols with arity 0 are constants. In addition to the constants and function symbols we assume the existence of an \(S\)-indexed family of sets \(X_{sort}\) where each \(X_{sort}\) contains an infinite number of variables of sort \(sort\). Any variable or constant is a term of the sort according to \(s\), and applying a function symbol to terms of the correct sort and number yields a term of the result sort of the function symbol. For instance, applying the function symbol \(succ: nat \rightarrow nat\) to the constant \(0: nat\), yields the term \((succ 0):nat\). Terms denote individuals.

The syntax of formulas is given in fig.C.1. The symbol \(x\) is a variable, \(t\) is a term, and \(r\) is some relation symbol in \(R\). We assume sorts are properly used.
APPENDIX C. MANY-SORTED FIRST ORDER LOGIC

\[ \phi ::= r(t_1, \ldots, t_n) \]
\[ \neg(\phi) \]
\[ \phi_1 \land \phi_2 \]
\[ \phi_1 \lor \phi_2 \]
\[ \phi_1 \implies \phi_2 \]
\[ \forall x : \text{sort}. \phi \]
\[ \exists x : \text{sort}. \phi \]
\[ t_1 =_{\text{sort}} t_2 \]
\[ \text{true} \]
\[ \text{false} \]

Figure C.1. First order syntax

\[ \| v \|_V^T = \mathcal{V}(v) \]
\[ \| c \|_V^T = m(c) \]
\[ \| f(t_1, \ldots, t_n) \|_V^T = m(f)(\| t_1 \|_V^T, \ldots, \| t_n \|_V^T) \]

Figure C.2. First order term interpretation

C.2 Semantics

The semantics of formulas and the denotations of the terms is given in relation to a structure, given as a tuple \( S = (U, m, \rho) \), and an assignment \( \mathcal{V} \), a function from variables to individuals. Here, \( U \) is the universe, or set of actual individuals the terms refer to. The symbol \( \rho \) is a function, \( \rho : S \rightarrow \mathcal{P}(U) \), which partitions the universe into disjunct subsets.

We call the subset belonging to a particular sort the domain of the sort. The symbol \( m \) is a function that assigns an element of \( U \) of appropriate sort to each constant in the vocabulary; a function of appropriate arity and sorts \( f : U_1 \times \ldots \times U_n \rightarrow U_{\text{result}} \) to each function symbol; and a subset of appropriate tuples from \( U_1 \times \ldots \times U_n \) to each relation symbol.

Given a structure \( S \) over a vocabulary, and an assignment \( \mathcal{V} \), we can give a recursive definition of what element of \( U \) each term denotes, according to fig.C.2. Here, the symbol \( v \) is some variable, \( c \) is a constant and \( f \) is some function symbol. The purpose of the assignment \( \mathcal{V} \) is to provide a denotation for free variables. We assume sorts are properly used.

The semantics of a formula, whether it is true or false, is given in relation to a structure \( S \) and an assignment \( \mathcal{V} \) in fig.C.3.

Given the semantics, we now introduce a few common notions. In the following, let \( \Gamma \) be a set of formulas and \( \phi \) any single formula.

If \( \| \phi \|_V^T = \text{true} \) under a given structure \( S \), we say \( \mathcal{V} \) satisfies \( \phi \) under \( S \) and write \( \mathcal{V} \models_S \phi \). Otherwise we write \( \mathcal{V} \not\models_S \phi \).

If \( \mathcal{V} \models_S \phi \) for any \( \mathcal{V} \), we say \( S \) is a model of \( \phi \) and write \( \models_S \phi \). Similarly, if \( \models \varphi \) for every \( \varphi \in \Gamma \), \( S \) is a model of \( \Gamma \).

\[ ^1 \text{Note that } \mathcal{V} \text{ does not have any influence if } \phi \text{ is closed, contains no free variables.} \]
C.2. SEMANTICS

$$\llbracket t_1, \ldots, t_n \rrbracket_V^F = (||t_1||_V^T, \ldots, ||t_n||_V^T) \in m(r)$$

$$\llbracket \neg(\phi) \rrbracket_V^F = \text{true iff } \llbracket \phi \rrbracket_V^F = \text{false}$$

$$\llbracket \phi_1 \land \phi_2 \rrbracket_V^F = \llbracket \phi_1 \rrbracket_V^F \land \llbracket \phi_2 \rrbracket_V^F$$

$$\llbracket \phi_1 \lor \phi_2 \rrbracket_V^F = \llbracket \phi_1 \rrbracket_V^F \lor \llbracket \phi_2 \rrbracket_V^F$$

$$\llbracket \phi_1 \implies \phi_2 \rrbracket_V^F = \llbracket \phi_2 \rrbracket_V^F \lor \neg(\llbracket \phi_1 \rrbracket_V^F)$$

$$\llbracket t_1 = \text{sort} \ t_2 \rrbracket_V^F = ||t_1||_V^T = ||t_2||_V^T$$

$$\llbracket \text{true} \rrbracket_V^F = \text{true}$$

$$\llbracket \text{false} \rrbracket_V^F = \text{false}$$

$$\llbracket \forall x : \text{sort} . \phi \rrbracket_V^F = \text{true iff for each } d \in \rho(\text{sort}) \ llbracket \phi \rrbracket_{V[\neg \to, d]}^F = \text{true}$$

$$\llbracket \exists x : \text{sort} . \phi \rrbracket_V^F = \text{true iff for some } d \in \rho(\text{sort}) \ llbracket \phi \rrbracket_{V[\neg \to, d]}^F = \text{true}$$

Figure C.3. First order formula semantics

If $\models_S \phi$ regardless of $S$ we say $\phi$ is a tautology, is valid, or is logically true and write $\models \phi$. Dually, if $\forall V \models_S \phi$ regardless of $V$ and $S$, $\phi$ is a contradiction.

We can now define semantic consequence.

If whenever for each $\varphi \in \Gamma$, $\forall V \models_S \varphi$ we also have $\forall V \models_S \phi$, we say the conclusion $\phi$ follows from the hypotheses $\Gamma$ in $S$ and write $\Gamma \models_S \phi$. If this is true regardless of structure, we say $\phi$ follows from, or is a semantic consequence of, $\Gamma$, and write $\Gamma \models \phi$. 
Appendix D

First order μ-calculus

D.1 Semantics

The semantics of a μ-calculus formula, given in fig.D.1, is a member of the set $\text{prop} \equiv ([t], \emptyset)$ (representing true and false) or a function that, when all arguments are supplied, returns a member of $\text{prop}$. The semantics is given in relation to a function $\kappa$ mapping free term– and predicate variables to terms and appropriate functions respectively. $\kappa$ is a variable that ranges over ordinals.

Given the semantics we can define semantic consequence similar to the first order case. In the following, let $\Gamma$ and $\Delta$ be sets of formulas, and $\phi$ be a single formula. Say $\rho$ satisfies $\phi$ and write $\rho \models \phi$ iff $\llbracket \phi \rrbracket_\rho = \{tt\}$. Say $\phi$ is a semantic consequence of $\Gamma$ and write $\Gamma \models \phi$ if whenever $\rho \models \phi$ for all $\gamma \in \Gamma$, it is also the case that $\rho \models \phi$. We say $\Delta$ is a semantical consequence of $\Gamma$ and write $\Gamma \models \Delta$ iff whenever all $\gamma \in \Gamma$ are satisfied by a valuation $\rho$, at least one $\delta \in \Delta$ is also satisfied by $\rho$. Note that we do not require all formulas of $\Delta$ to be satisfied by $\rho$.

The semantics of the fixed point formulas deserves some elaboration. The meaning of a least fixed point formula is the least fixed point of a corresponding function over functions.

Consider the following fixed point formula.

$$ \phi \equiv \mu U : s_\phi.\phi(U) $$

We have $\phi : s_\phi, \phi : s_\phi$ and $U : s_\phi$, meaning all of these should be applied to a sequence of terms $t : s_1 \ldots s_n$ such that $s_1 \rightarrow \ldots s_n \rightarrow \text{prop} = s_\phi$.

Now, according to the semantics, both $\llbracket \phi(U) \rrbracket_\rho$ and $\llbracket U \rrbracket_\rho$ are curried functions of the following general shape:

$$ f_\phi : s_1 \rightarrow \ldots s_n \rightarrow \text{prop} $$

Furthermore, $\llbracket \phi(U) \rrbracket_\rho$ depends on the $\llbracket U \rrbracket_\rho$ as given by $\rho$. The function whose least fixed point we choose as $\llbracket \mu U : s_\phi.\phi(U) \rrbracket_\rho$ is:

$$ f_\phi(A) : [s_1 \rightarrow \ldots s_n \rightarrow \text{prop}] \rightarrow [s_1 \rightarrow \ldots s_n \rightarrow \text{prop}] \equiv \llbracket \phi(U) \rrbracket_\rho[A/U] $$

meaning for $U$
\textbf{APPENDIX D. FIRST ORDER $\mu$-CALCULUS}

\[
\begin{align*}
\|t_1 = t_2\|_\rho & \triangleq \text{if } \rho(t_1) = \rho(t_2) \text{ then } \{tt\} \text{ else } \emptyset \\
\|\phi_1 \lor \phi_2\|_\rho & \triangleq \|\phi_1\|_\rho \cup \|\phi_2\|_\rho \\
\|\neg \phi\|_\rho & \triangleq \{tt\} - \|\phi\|_\rho \\
\|\exists x : \text{sort}_x, \phi\|_\rho & \triangleq \bigcup_{\forall\text{terms}(\text{sort}_x)} \|\phi\|_{[v/x]} \\
\|\lambda x : \text{sort}_x, \phi\|_\rho & \triangleq \lambda x' : \text{sort}_x, \|\phi\|_{[x'/x]} \\
\|\mu U : s_\phi, \phi\|_\rho & \triangleq \bigcup \{\|\mu U : s_\phi, \phi\|_{[\beta/\xi]} | \beta \in \text{ord}\} \\
\|\mu U : s_\phi, \phi\|_{\rho} & \triangleq \begin{cases} \\
\lambda x_1 : s_1, \ldots, \lambda x_n : s_n, \emptyset & \text{if } s_\phi = s_1 \rightarrow \ldots \rightarrow s_n \rightarrow \text{prop} \\
\emptyset & \text{if } \rho(k) = \beta + 1 \\
\bigcup \{\|\mu U : s_\phi, \phi\|_{[\beta/\xi]} | \beta < \rho(k)\} & \text{if } \rho(k) \text{ is a limit ordinal} \\
\end{cases} \\
\|U\|_\rho & \triangleq \rho(U)
\end{align*}
\]

\textbf{Figure D.1. Semantics of the $\mu$-calculus}

We know such a fixed point exists because the constraint of $U$ occurring under an even number of negations guarantees that $f_\phi$ is a monotone function over the lattice

$$\langle [s_1 \rightarrow \ldots \rightarrow s_n \rightarrow \text{prop}], \subseteq \rangle$$

where $\subseteq$ is defined as follows.

For $f_1, f_2 \in \text{prop}$ (remember $\emptyset \subseteq \{tt\}$):

$$f_1 \subseteq f_2 \iff f_1 \subseteq f_2$$

For $f_1, f_2 \in [s \rightarrow s_\phi]$:

$$f_1 \subseteq f_2 \iff \forall t : s. f_1(t) \subseteq f_2(t)$$

Here, $f_2$ is possibly $\{tt\}$ for more arguments than $f_1$, so intuitively $\subseteq$ orders the functions according to completely they cover the true term applications.

Now finally, for an arbitrary set $F \subseteq [s \rightarrow s_\phi]$:

$$\begin{align*}
\bigcup F & \triangleq \lambda t : s. \bigcup_{f \in F} f(t) \\
\bigcap F & \triangleq \lambda t : s. \bigcap_{f \in F} f(t)
\end{align*}$$

\textbf{D.2 A proof system}

The proof rules are presented below in groups together with some discussion. $\Gamma$ and $\Delta$ are multisets of formulas, and $\phi$ a single formula.

In the set of proof rules in fig.D.2, ID is an axiom. Obviously, if $\phi$ is valid on the right hand side, it must also be valid on the left hand side.
D.2. A PROOF SYSTEM

![Figure D.2. Structural rules](image)

By reading the WEAKEN-rules top-down, if \( \Gamma \vdash \Delta \) is valid in the premise, then adding a formula on either side cannot make the resulting sequent invalid. By reading them bottom-up as we do when constructing the proof, if we are to prove the sequent in the consequence, then removing a formula from either side does not make our task easier. That is, by removing the formula we cannot suddenly deduce more than before, in particular we cannot deduce a sequent which is not valid.

The CUT-rule is special in that when read bottom-up, it possibly introduces a new formula. In a sense, it is the opposite of the WEAKEN-rules. If we are to prove \( \Gamma \vdash \Delta \) but would need \( \phi \) as an extra premise, the CUT-rule allows us to introduce \( \phi \) on the left hand side provided we can also prove \( \phi \) (or \( \Delta \), but in that case there is no point really) is already a consequence of the premises \( \Delta \). The CUT-rule will be used later on as a building block for the TERMCUT-rule.

![Figure D.3. Logical rules](image)

The set of proof rules in fig.D.3 handle the usual connectives and abstractions. For the \( \exists \) rule we have a side condition. Looking back at the semantics, we see that an exists-formula is true when some choice of \( x \) makes the body of the formula \( t[t] \). Our interest here lies in proving the exists-formula false, that is, to show that all choices of \( x \) make the body of the formula false, \( \Theta \). That would seem to mean we need one proof rule premise for each choice of \( x \), which clearly is not practical. Instead, we choose \( x_0 \) as one canonical representative for all choices of \( x \), and validate this single premise by disproving the formula. For this to work, the \( x_0 \) must not occur in the conclusion sequent, i.e. are not allowed to have any other knowledge about \( x_0 \). This is what we mean by saying \( x_0 \) must be fresh. We call the introduced \( x_0 \) a fresh representative in various places in the paper.

Note that fresh representatives are also introduced by the symmetrical \( \forall \) rule.

The proof rules in fig.D.4 handle the equality symbol: it is reflexive, symmetric and transitive and a congruence, meaning we may substitute equals for equals (using the SUBST-
rule). The transitivity and symmetry rules can be derived, meaning that the same results can be achieved by using a series of other proof rules.

\[
\begin{align*}
\text{REFL:} & \quad \Gamma \vdash t =_s t, \Delta \\
\text{SYMM}_{L}: & \quad \Gamma, t_2 =_s t_1 \vdash t_1 =_s t_2, \Delta \\
\text{TRANS}_{L}: & \quad \Gamma, t_1 =_s t_2, t_2 =_s t_3 \vdash \Gamma, t_1 =_s t_3, \Delta \\
\text{SUBST:} & \quad \Gamma[t_2/x] \vdash \Delta[t_2/x] \\
\text{SYMM}_{R}: & \quad \Gamma, t_1 =_s t_2, \Delta \\
\text{TRANS}_{R}: & \quad \Gamma \vdash t_1 =_s t_2, t_2 =_s t_3, \Delta
\end{align*}
\]

\[\text{Figure D.4. Equality rules}\]

The proof rules in fig.D.5 are for sorts where syntactically different terms are never equal. This is true for all sorts we use in this thesis.

\[
\begin{align*}
\text{CEQ}_{L}: & \quad \Gamma, t_1 =_{s_1} t'_1 \ldots t_n =_{s_n} t'_n \vdash \Delta \\
\text{CEQ}_{R}: & \quad \Gamma \vdash op(t_1, \ldots, t_n) = op(t'_1, \ldots, t'_n), \Delta \\
\text{CEQ}_{L}': & \quad \Gamma \vdash op(t_1, \ldots, t_n) = op(t'_1, \ldots, t'_n), \Delta \\
\text{CEQ}_{R}': & \quad \Gamma, t_1 =_{s_1} t'_1 \ldots t_n =_{s_n} t'_n \vdash \Delta \\
\end{align*}
\]

\[\text{Figure D.5. Equality rules for freely generated sorts}\]

\[
\begin{align*}
\text{FALSE}_{L}: & \quad \Gamma, \text{false} \vdash \Delta \\
\text{CONTRACT}_{L}: & \quad \Gamma, \phi, \phi \vdash \Delta \\
\text{CONTRACT}_{R}: & \quad \Gamma \vdash \phi, \phi, \Delta \\
\text{\leftarrow}_{L}: & \quad \Gamma, \phi \vdash \Delta \\
\text{\leftarrow}_{R}: & \quad \Gamma \vdash \phi \vdash \Delta \\
\end{align*}
\]

\[\text{Figure D.6. Derived proof rules}\]

The proof rules in fig.D.6 are for the abbreviated operators. All of them are derived.

\[
\begin{align*}
\text{TERMCUT:} & \quad \Gamma \vdash \lambda t : \psi, \Delta, \Gamma, x : \psi \vdash t : \phi, \Delta \\
\end{align*}
\]

\[\text{Figure D.7. TERMCUT rule for compositive reasoning}\]
D.2. A PROOF SYSTEM

The proof rule TERMCUT in fig. D.7 is also a derived proof rule. It is of great use for compositional reasoning about properties of processes, where a property of the process as a whole depends on properties of subprocesses. For instance, think of \( t(t/x) \) as some process we wish to prove satisfies \( \phi \). The proof rule allows us to use the fact that some subprocess \( t' \) satisfies some property \( \psi \) as a “leverage” in proving our original goal.

We now turn to the fixed point operators. The proof rules in fig. D.8 follow the more constructive definition of fixed points as limits of a chain of approximations. In general we cannot “compute” this chain by a tedious series of unfoldings to see whether some property holds or not. Instead we employ approximation ordinals whose purpose is to provide a progress measure towards the actual fixed point. These approximation ordinals are decremented by the proof rules each time we unfold an approximation. If by approximation and unfolding we can make essentially the same fixed point formula reappear but with a suitable change in the approximation ordinal, we may be able to conclude that the fixed point formula holds and finish, discharge, that branch of the proof tree. This judgment must typically take other parts of the proof into account and cannot be formulated as a proof rule acting on only one sequent. We will not present the details of this discharge mechanism, but refer the reader to previously mentioned papers.

Complementary rules for greatest fixed points are presented in fig. D.9. In addition, the rules in fig. D.10 are needed for reasoning about ordinals.

\[
\text{APPROX}_L: \quad \frac{\Gamma, ((\mu U : s_{\phi \psi})^\kappa) \Gamma_1 \ldots \Gamma_n \vdash \Delta}{\Gamma, (\mu U : s_{\phi \psi}) \Gamma_1 \ldots \Gamma_n \vdash \Delta} \quad \uparrow \\
\mu \text{UNFOLD}_1_L: \quad \frac{\Gamma, (\phi(\mu U : s_{\phi \psi}(U))) \Gamma_1 \ldots \Gamma_n \vdash \Delta}{\Gamma, (\mu U : s_{\phi \psi}) \Gamma_1 \ldots \Gamma_n \vdash \Delta} \\
\mu \text{UNFOLD}_1_R: \quad \frac{\Gamma \vdash (\phi(\mu U : s_{\phi \psi}(U))) \Gamma_1 \ldots \Gamma_n, \Delta}{\Gamma \vdash (\mu U : s_{\phi \psi}) \Gamma_1 \ldots \Gamma_n, \Delta} \\
\mu \text{UNFOLD}_2_L: \quad \frac{\Gamma, (\phi((\mu U : s_{\phi \psi})^\kappa)(U))) \Gamma_1 \ldots \Gamma_n, \kappa < \kappa', \Delta}{\Gamma, (((\mu U : s_{\phi \psi})^\kappa)(U))) \Gamma_1 \ldots \Gamma_n, \Delta} \quad \updownarrow \\
\mu \text{UNFOLD}_3_R: \quad \frac{\Gamma \vdash (\phi((\mu U : s_{\phi \psi})^\kappa)(U))) \Gamma_1 \ldots \Gamma_n, \Delta}{\Gamma \vdash ((\mu U : s_{\phi \psi})^\kappa)(U))) \Gamma_1 \ldots \Gamma_n, \Delta} \quad \updownarrow \kappa \text{ must be fresh} \\
\updownarrow \kappa' \text{ must be fresh}
\]

Figure D.8. Least fixed point proof rules
APPENDIX D. FIRST ORDER μ-CALCULUS

\[ \text{APPROX}_R: \quad \Gamma \vdash (\forall U : s_\theta. \phi)^k t_1 \ldots t_n, \Delta \]
\[ \text{vUNFOLD1}_L: \quad \Gamma, (\forall U : s_\theta. \phi) t_1 \ldots t_n \vdash \Delta \]
\[ \text{vUNFOLD1}_R: \quad \Gamma \vdash (\forall U : s_\theta. \phi) t_1 \ldots t_n, \Delta \]
\[ \text{vUNFOLD2}_R: \quad \Gamma, \kappa < \kappa' \vdash (\phi((\forall U : s_\theta. \phi)^k U)) t_1 \ldots t_n, \Delta \]
\[ \text{vUNFOLD3}_L: \quad \Gamma, ((\forall U : s_\theta. \phi)^k U) t_1 \ldots t_n \vdash \Delta \]
\[ \Gamma, (\forall U : s_\theta. \phi) t_1 \ldots t_n \vdash \Delta \]

\[ \text{†} \kappa \text{ must be fresh} \]
\[ \text{‡} \kappa' \text{ must be fresh} \]

Figure D.9. Greatest fixed point proof rules

\[ \text{IDMON1:} \quad \Gamma \vdash \kappa < \kappa', \Delta \]
\[ \Gamma, ((\forall U : s_\theta. \phi)^k) t_1 \ldots t_n \vdash ((\forall U : s_\theta. \phi)^k) t_1 \ldots t_n, \Delta \]

\[ \text{IDMON2:} \quad \Gamma \vdash \kappa < \kappa', \Delta \]
\[ \Gamma, ((\forall U : s_\theta. \phi)^k) t_1 \ldots t_n \vdash ((\forall U : s_\theta. \phi)^k) t_1 \ldots t_n, \Delta \]

\[ \text{ORDTRANS:} \quad \Gamma, \kappa < \kappa', \kappa' < \kappa'', \kappa < \kappa'' \vdash \Delta \]
\[ \Gamma, \kappa < \kappa', \kappa' < \kappa'' \vdash \Delta \]

Figure D.10. Ordinal transitivity and monotonicity of approximations

72
Appendix E

Implementation excerpts

The implementation consists of three main theories. First the $\pi$-calculus theory with the
definition of process terms, the various syntactic operations and the transition relation. Then
a supporting set theory, which represents sets of natural numbers as lists. The principal
operation is to determine whether a natural number occurs in such a set. The set theory is
for example used for some side conditions of the transition rules. The final main theory is a
theory for simple explicit functions over natural numbers. We represent such a function as
a list of natural number-pairs. The principal operation is to evaluate a natural number with
respect to a function. We use this function theory in the translated MPW-formulas, and in
the recursive definition of alpha equivalence.

In the following sections we give excerpts from each of these theories. Since some
tactics have been parameterised, we have chosen some simple instantiation. Some parts of
the theory files have been left out to avoid bloat.

The complete implementation is available on request.

E.1 Set

The following is an excerpt from the theory file with the signature for sets, and the named
formula for set membership.

```
THEORY Set
REQUIRES Utility, Simplify;
DEFINITIONS

DATATYPES
  elementtype = nat;
  set = nat list
END

PREDICATES
  member : elementtype --> set --> prop <=
  \E:elementtype . \S:set .
  cases S of
    [] : set => ff
    | [EE | Ss] => E = EE or member E Ss
  end
end;
```
An example of a right hand side tactic (from “tactics.sml”) for handling set membership formulas is as follows.

```sml
fun set_member_r frmPos =
  t_fix
  frmPos
  (fn pos =>
   {fn recurse =>
    t-compose_l
    [beta_reduce_r pos,
     cases_r pos,
     or_r pos,
     t-orelse_l
     [exteq_r pos,
      t-compose_l
      [weak_r pos,
       recurse pos
     ]]
   ]
  })

```

`t_fix` is a tactical for creating recursive tactics. In this example, read `recurse pos` as looping from the `recurse =>`.

### E.2 Function

The following is an excerpt from the theory file with the signature for functions, and the named formula for evaluating a function term.

```sml
THEORY Function
REQUIRES Utility, Simplify;
DEFINITIONS

DATATYPES
domain = nat;
range = nat;
function = (domain*range) list
END

PREDICATES
eval : function --> domain --> range --> prop <=
  \F:function . \X:domain . \Y:range .
  (exists F' : function . F = [(X, Y) | F']) or
  (exists F' : function . exists X' : domain . exists Y' : range .

```
E.2. FUNCTION

\[ F = \{ (X', Y') \mid F' \text{ and } (\neg X' = X \text{ and } \text{eval } F' X Y) \} \]

end
END

SML FROM "tactics.sml"

END

The corresponding right hand side tactic is as follows.

fun eval_r frmPos =
  t_fix
    frmPos
  (fn pos =>
    (fn recurse =>
      t_compose_l
      [ beta_reduce_r pos,
        or_r pos,
        t_orelse_l
        [ t_compose_l
          [ weak_r (pos+1),
            exists_r NONE pos,
            t_compose (eq_flat_decomp_r pos)
          [ t_compose (eq_flat_decomp_r pos)
            [ t_orelse_l [exeq_r pos],
              t_orelse_l [exeq_r pos]
          ]},
            exeq_r pos
          ]],
        t_compose_l
        [ weak_r pos,
          exists_r NONE pos,
          exists_r NONE pos,
          exists_r NONE pos,
          t_compose
            (and_r pos)
          [ exeq_r pos,
            t_compose
              (and_r pos)
            [ t_compose_l
              [ not_r pos,
                apply_to_last_formula_l eq_flat_elim_l
              ],
                recurse pos
            ]
          ]
        ]
      ]

  )
)
E.3 \(\pi\)-calculus

We have already seen excerpts of the signature defining \(\pi\)-calculus process terms and action terms, and parts of the formula definition of the transition relation in chapter 3. Below we reproduce a few of the right and left hand side transition lemmas.

/* right hand side */

LEMMAl_out
DECLARE P : process, X : name, Y : name IN
|- trans sum(seq(out(X,Y), P)) fo(X,Y) P
END

LEMMAl_match
DECLARE P0 : prefix, P : process, A : action, P' : process IN
trans sum(seq(P0, P)) A P'
|- forall X : name . trans sum(seq(match(X, X, P0), P)) A P'
END

LEMMAl_res
DECLARE P : process, A : action, P' : process, Z : name IN
trans P A P', (not in_action_names Z A)
|- trans new(Z,P) A new(Z,P')
END

LEMMAl_comm_l
DECLARE P : process, P' : process, Q : process, Q' : process IN
(exists X : name .
  exists Y : name .
  exists Z : name .
  exists Q0 : process .
  trans P fo(X,Y) P' and trans Q bi(X,Z) Q' and subst Q' Z Y Q''
)
|- trans compose(P,Q) silent() compose(P',Q'')
END

LEMMAl_alpha
DECLARE P : process, A : action, Q' : process IN
(exists Q : process . alpha_eq P Q and trans Q A Q')
|- trans P A Q'
END

/* left hand side */

LEMMAc_out
DECLARE X : name, Y : name, P : process, A : action, P' : process IN
trans sum(seq(out(X,Y), P)) A P'
|- (A = fo(X,Y) and
  alpha_eq P P')
END

LEMMAc_inp
DECLARE X : name, Y : name, P : process, A : action, P' : process IN
trans sum(seq(inp(X,Y), P)) A P'
|-
E.3. $\pi$-CALCULUS

{exists $Z$ : name .
A = bi($X,Z$) and
alpha_eq sum(seq(inp($X,Y$),P)) sum(seq(inp($X,Z$),P'))}
)
END

LEMMA c_compose
DECLARE P : process, Q : process, A : action, PQ' : process IN
trans compose(P,Q) A PQ'
|- {exists $P'$ : process .
extists QA : process .
PQ' = compose($P'$, QA) and
trans P A P' and
alpha_eq Q QA and
{exists BAN : set . exists FNQA : set .
bound_action_names A BAN and
free_process_names QA FNQA and
intersection BAN FNQA {} : set }
),
),
... symmetrical

/* alpha conversion can take place in trans for P/Q */
(A=silent() and
{exists $P'$ : process .
extists Q' : process .
extists $X$ : name .
extists $Y$ : name .
extists $Z$ : name .
extists $Q''$ : process .
PQ' = compose($P'$, Q') and
trans P $fo(X,Y)$ $P'$ and
trans Q $bi(X,Z)$ $Q''$ and
subst $Q''$ $Z$ $Y$ $Q'$
})
),
... symmetrical

/* alpha conversion can take place in trans for P/Q */
(A=silent() and
{exists $P'$ : process .
extists Q' : process .
extists $X$ : name .
extists $Z$ : name .
PQ' = new($Z$, compose($P'$, Q')) and
trans P $bo(X,Z)$ $P'$ and
trans Q $bi(X,Z)$ $Q'$
})
),
... symmetrical
END
Appendix F

A bug in VCPT

We entered the following proof goal in one of our theory files.

```
GUESS bug
|- exists F : function . F = [(0,0), [] : function]
END
```

Here, the second comma is invalid for the term to be a proper function term, and should be a `|` symbol.

When starting VCPT and attempting to prove this goal, the initial sequent now looks as follows.

```
|- (1) exists F:function. F=\((0,0)|][][[]])
```

The term `\((0,0)|][][[]])` is not a valid function term, so from our understanding, this proof ought to fail. However, the proof can be completed by applying `exists_r NONE 1`, followed by `exteq_r 1`. Even with the global flag to generate type checking goals, the proof can be completed.

We have not attempted to find the root cause of this bug we but suspect it can be found in the parsing or type checking code.