Implementing Typed Psi-calculi

Amin Khorsandiaghai

Institutionen för informationsteknologi

Department of Information Technology
Abstract

Implementing Typed Psi-calculi

Amin Khorsandiaghai

In this thesis project we focus on the implementation of a type system in a tool called “Psi-calculi Workbench”. Psi-calculi is an attempt to generalize a family of process calculi based on a parametric approach. Instantiating parameters of the psi-calculi framework (namely: data, conditions and assertions) results in a psi-calculus. The implementation of the type system is based on an existing specification with some extensions which we add to it. The original idea behind the psi-calculi type system is to keep it as generic as possible, with minimum dependency to the specific psi instances. Due to this requirement imposed by the theory, we try to keep the implementation of the type system as generic as possible. This can be achieved in a rather straightforward way by virtue of Standard ML module system.

In order to evaluate the result of our type system implementation, we present the instantiation of two typed psi-calculus examples using the typed version of the Psi-calculi Workbench.
## Contents

1 Introduction 6  
 1.1 Psi-calculi 7  
 1.2 Type systems 7  
 1.3 Contribution 8  
 1.4 Outline 9  

2 Background theory 10  
 2.1 Nominal logic 10  
 2.2 Type systems 12  

3 Typed psi-calculi 16  
 3.1 Typed syntax 19  
 3.2 Typed symbolic operational semantics 19  
 3.3 Psi-calculi type system 21  
    3.3.1 Type environment 23  
    3.3.2 Type judgements and type rules 24  
 3.4 Process constants 25  
 3.5 Implementations 30  

4 Typed Psi-calculi Workbench 33  
 4.1 Tool 33  
    4.1.1 Typed syntax 33  
    4.1.2 Extended command interpreter 34  
 4.2 Implementation of Psi-calculus instances 36  
    4.2.1 Pi-calculus instance 36  
    4.2.2 Sample run sessions for pi-calculus instance 43  
    4.2.3 Distributed Pi-calculus instance 51  
    4.2.4 Sample run sessions for Dpi-calculus instance 58  

5 Conclusion 63  
 5.1 Future work 63  
 5.2 Related work 63  
 5.3 Acknowledgments 64  

References 66  

A Type-checker implementation 69  

B Psi Workbench grammar (with types) 76  
  B.1 Command grammar 76  
  B.2 Agent grammar 76
1 Introduction

Concurrent software and systems have been around for more than four decades. They are everywhere around us, and have become an inseparable part of our life. Due to this matter of importance, the need to develop error-free and correct concurrent software and systems have become a necessity. Formal reasoning is a method that computer scientists use in order to study the software systems and to see if the system is actually following its specification and does what it must. Researchers build a model which abstracts away unnecessary details of the system and provides them a facility to investigate the system in a very efficient way. Process calculi is such a formal model for describing and reasoning about communicating concurrent systems and parallel/distributed computation.

There exist a broad range of process calculi. The Communicating Sequential Processes (CSP) [15], the Calculus of Communicating Systems (CCS) [23], the Algebra of Communicating Processes (ACP) [7], and the pi-calculus [26] are examples of the formal models that try to describe concurrent communicating systems. CSP developed by C.A.R Hoare in 1978 is a formal language which describes the patterns of interaction between different processes in a concurrent system [15]. In this work, the input, output and concurrency are considered as primitives of programming [15]. CCS introduced by Robin Milner around 1980 and is a process calculus useful for evaluating correctness properties (such as absence of deadlock or livelock) of a concurrent system [23]. ACP initially developed by Jan Bergstra and Jan Willem Klop in 1982 and its focus is on algebraic reasoning about concurrent systems [7].

The pi-calculus which has been developed as a result of the continuation of the work on CCS, is a beautifully simple yet powerfully expressive process calculus that describes the mobile concurrent systems [27]. The concept of mobility in concurrent systems means that the configuration of the system might change during the computation; in pi-calculus for example the communication links that connect processes are mobile. The pi-calculus has a multitude of extensions where higher-level data structures and operations on them are given as primitives [5]. Example of such extensions are the applied pi-calculus [4] by Abadi and Fournet and the spi-calculus [3] by Abadi and Gordon. According to [5], the former focuses on cryptographic primitives, and the latter extends the pi-calculus with dynamically scoped names (aliases) for data. Bengtson et al. [5] have shown that these instances, along with many other proposed extensions of the pi-calculus, can be formulated using a parametric framework called Psi-calculi.
1.1 Psi-calculi

The psi-calculi is a parametric framework extending the pi-calculus with nominal data types for data structures and for logical assertions representing facts about data [5]. Extensions of the pi-calculus such as applied pi-calculus [4], the spi-calculus [3], the fusion calculus [38], the concurrent constraint pi-calculus [8] and calculi with polyadic communication channels or pattern matching [9] can be formulated as psi-calculi [5]. The general algebraic properties of psi-calculi have been proof-checked in the interactive proof assistant Isabelle [6]. Recently, a tool based on the psi-calculi has been implemented to provides a facility for users to instantiate and investigate the various instances and extensions of pi-calculus [13]. The tool is called Psi-calculi Workbench. In order to make use of the workbench, the user must provide three nominal datatypes and four equivariant operators defined on those datatypes in an appropriate way. In this way, the framework is able to host a wide range of process calculi. Later in Section 3 we discuss the psi-calculi framework (in a typed version) and its different parameters. In this thesis project we extend the implementation of the Psi-calculi Workbench, enhancing it with support for type systems.

1.2 Type systems

Type systems have been used for reasoning about the correctness properties of the process calculi. For example the type system established by Milner [26], for the pi-calculus [26], establishes the correct use of channels by checking that only names with the appropriate type can be communicated on channels (a channel here can be seen as a communication link which connects two agents). Many other type systems for the pi-calculus and its variants have been introduced. One example is the type system for pi-calculus by Sangiorgi and Walker [33]. They start by a Base-$\pi$ type system for CSS with value passing and then they incrementally add more complex type constructs to it, developing more intricate type systems for simply typed pi-calculus [17]. Another example of a type system for pi-calculus is the one described by Pierce and Sangiorgi in [29]. In this type system, the concepts of sub-typing and capability tags have been deployed in order to control the use of names as output and input channels. There have been works on the type systems for variants of the pi-calculus as well. The type system for distributed pi-calculus (D$\pi$) by Hennessy and Riely [32], for example, have been introduced to control the migration of the subprocesses between locations. For the spi-calculus by Abadi and Gordon [3] – a calculus intended for cryptographic protocols – there have been works on developing type systems in order to capture the properties of the cryptographic protocols, known as secrecy and authenticity. A type system by Abadi [1] and another one by Abadi and Blanchet [2] target the secrecy property of spi-calculus. Gordon
and Jeffery have developed a type system aiming for authenticity property of spi-calculus \cite{12}. The type system by Pierce and Sangiorgi \cite{29}, which we discussed above, also captures the authenticity property of cryptographic protocols.

The type systems mentioned above seem to be unrelated to each other and have been developed in response to different needs, but the point is that, they share certain characteristics: all of them classify processes as either well-typed or not. Therefore, all the above type systems have type judgments for the processes in the following form:

\[
E \vdash P
\]

This means that, the process \( P \) is well typed under the type environment \( E \) that keeps track of the types of free names in \( P \). Conversely, the terms in the calculi described above, are given the type \( T \) under the associating type systems, therefore the judgment for the term \( M \) under the type environment \( E \) is like:

\[
E \vdash M : T
\]

These shared characteristics among seemingly unrelated type systems, developed for pi-calculus and its variants, implies that it might be possible to define a generic type system which can capture all the type systems of the different process calculi \cite{17}. Hüttel shows in \cite{17} that it is possible to establish a general frame-work for type systems for the variants of the pi-calculus in a similar role that psi-calculi plays as a framework capturing the variants of pi-calculus; this is achieved by establishing a family of type systems for psi-calculi.

The type system specified by Hüttel generalizes the simply typed pi-calculus introduced by Sangiorgi and Walker \cite{33}. Within this generic type system, it is possible to capture many existing, apparently quite different type systems such as those we mentioned above. It is also possible to introduce new type systems under this framework.

1.3 Contribution

The purpose of this thesis work is to implement the type system of psi-calculi in the Psi-calculi Workbench \cite{13}. The implementation is according to the simple type system for the psi-calculi by Hüttel \cite{17}.

Some of the tasks of this thesis project are introducing types in different parts of the tool such as the syntax of the psi-calculus and its symbolic operational semantics as well as in the transition constraints. We also extend the set of commands available in the command interpreter of the tool; command for inserting names with their types into the type environment or command for type-checking a process etc.
Gutkovas [13] has extended the original syntax of the psi-calculus by adding another form of process to it. This additional agent form is called Invocation (process constant) and is present in the original syntax of pi-calculus. In this thesis work, we introduce a typing rule for the invocation process and give a proof of its subject reduction property.

1.4 Outline

The remaining parts of this text are outlined as follows:

In the Background theory chapter we describe the required theoretical background for our type system implementation. First we introduce the concept of Nominal logic (mainly the nominal sets) and its application in the psi-calculi theory. Then we discuss the background theory required by the user to get some idea about type systems and their application in computer science and software engineering in the second subsection of this section.

In the chapter Typed psi-calculi, first we go further into the design details of the type system for psi-calculi. We describe the theoretical aspects of the psi-calculi framework, its different parameters and formal syntax. Then we introduce the symbolic operational semantics of psi-calculi. Finally we discuss the typed version of the INVOCATION transition rule, the typing rule for the Invocation agent and give a proof for its correctness property.

In Typed Psi-calculi Workbench chapter we mainly focus on the procedure of implementing typed instances using the tool. We discuss the extensions made to the command interpreter and the new commands. We describe the process of implementing two typed instances of psi-calculi in the workbench in a step by step fashion, and give examples of running sessions for both instances.

Finally in the Conclusion chapter, we discuss similar works and projects related to our work and also sketch an outline of the possible extensions and future works regarding this thesis project.
2 Background theory

The theory section is intended to give the reader an insight about what is going to become in the upcoming chapters of the thesis and what are the theoretical backgrounds and requirements that this thesis job is based on. We start by giving the concepts from Nominal logic \cite{31} which has been used to define psi-calculus theory. We also give a general overview of the type systems and their use in the formal software development. These concepts are used in the next chapter, when we discuss about the typed psi-calculi theory.

2.1 Nominal logic

There has been efforts to make computers able to do mathematical proofs as a counterpart for the original pen and paper proofs which mathematicians normally do. Formal methods and theorem provers are examples of such efforts. Before introducing Nominal logic \cite{31}, there has been a gap between the techniques available for the machine checked proofs and the alternative pen and paper style. When conducting pen and paper proofs, one important thing that one must take care of is to avoid capture by binders when introducing free variables. This procedure is quite well understood and straight-forward in pen and paper proofs (though one must still be so careful); but when it comes to the machine assisted proof, the situation may not be always so straightforward. Considering the fact that computers are finite machine (regarding their memory size and computational capabilities), they are not able to check all the states (or possible values) of a variable when conducting a proof \cite{13}. The solution to this problem is to feed the proof assistant with well chosen representative for variables and then do the induction on them. But here another problem arises. The induction on the representatives some times is too weak, and therefore not suitable as a replacement for the pen and paper proofs.

To overcome this issue, Nominal logic was introduced. The rationale behind Nominal logic is to make the machine assisted proofs as similar to pen and paper proofs as possible \cite{36}. What is interesting about Nominal logic is that its underlying concepts are very well suited to be integrated into a programming language (functional languages in particular) and to help making the task of construction of syntactical structures easier \cite{34}.

Now we are going to see the formal definitions of the Nominal logic theory. These definitions are required by the implementation of the type system of psi-calculi. The following definitions are adapted from \cite{19} Section 3.1.1 (except Definition 1, which is from \cite{13} Section 2.1).

\textbf{Definition 1.} (Atoms (or Names)) A countably infinite set $\mathcal{N}$. We Range over it by $a, b, \ldots$. 
The set of atoms must be countable.

**Definition 2** (Atom swapping and permutation). A name swapping operation on expression \( E \) is written as \((a \ b) \cdot E\) and is a simultaneous renaming of \(a\) by \(b\) and \(b\) by \(a\) in \(E\). This operation must satisfy the following requirements

\[
(a \ a) \cdot E = E \\
(a \ b) \cdot (a \ b) \cdot E = E \\
(a \ b) \cdot (c \ d) \cdot E = ((a \ b) \cdot c \ (a \ b) \cdot d) \cdot (a \ b) \cdot E
\]

where \((a \ b) \cdot c \begin{align*}
&= \begin{cases} 
  a & \text{if } c = b \\
  b & \text{if } c = a \\
  c & \text{otherwise}
\end{cases} \\
&\text{and similarly for } (a \ b) \cdot d.
\end{align*}\]

A permutation \(\rho\) on some expression \(E\) is a sequence of name swapping on \(E\), and we write it as \(\rho \cdot E\).

**Definition 3** (Support). The support of an expression \(E\) is the least set of names \(A\) such that \((a \ b) \cdot E = E\) for all \(a, b \notin A\). We write the support of \(E\) as \(n(E)\).

In fact, the set of atoms which affect the term when swapped constitutes the support of that term.

The next property is known as the freshness property.

**Definition 4** (Freshness). Let \(a\) be an atom and \(E\) be an expression, then \(a\) is said to be fresh in \(E\) if

\[a \# E \overset{\text{def}}{=} a \notin n(E)\]

**Definition 5** (Nominal set). A nominal set \(A\) is a set equipped with name swapping operations. We require that for each \(E \in A\) the set \(n(E)\) is finite.

**Definition 6** (Equivariant functions). A function \(f\) on nominal sets is equivariant if it has the following property

\[
\rho \cdot f(x_1, \ldots, x_n) = f(\rho \cdot x_1, \ldots, \rho \cdot x_n)
\]

for all permutations \(\rho\).

Intuitively, the equivariance property means that the equality is preserved under the atom swapping.

**Definition 7** (Substitution). A substitution is an equivariant function which satisfies the following requirements
• Freshness:
  \( \bar{x} \subseteq n(E) \land a \# E[\bar{x} := \bar{E}'] \Rightarrow a \# \bar{E}' \)

• \( \alpha \)-equivalence:
  \( \rho \subseteq \bar{x} \times (\rho \cdot \bar{x}) \land (\rho \cdot \bar{x}) \# E \Rightarrow E[\bar{x} := \bar{E}'] = (\rho \cdot E)[(\rho \cdot \bar{x}) := \bar{E}'] \)

The substitution of a sequence of expressions \( \bar{E} \) for an equally long sequence of names \( \bar{a} \) in an expression \( E' \) is written as

\[ E'[\bar{a} := \bar{E}] \]

We use \( \sigma \) to range over substitutions.

Finally, we give the definition of the nominal datatype.

**Definition 8** (Nominal datatype). A nominal datatype is a nominal set with a set of equivariant functions defined on it, including a substitution function.

The psi-calculus theory is based on Nominal sets [19]. Most of the psi-calculus meta-theory has been formalized and machine-checked [6] using the Nominal Isabelle [36] theorem prover.

### 2.2 Type systems

In this section we give a gentle introduction into types and type systems. What is a type system? Why it is useful? How we can implement or deploy a type system to develop correct software? These are some of the questions we try to answer in the following paragraphs. First we discuss the role that types and type systems play in the process of programming language design and development. Later we give the theoretical definitions and background related to the type systems and their implementation.

Type systems are a mechanism deployed by language designers to classify the expressions of a language. There are several reasons that a type system is useful, notably:

- to detect programming errors statically
- to extract information that is useful for reasoning about the behavior of programs
- to improve the efficiency of code generated by a compiler
- to make programs easier to understand [33].

The items listed above, is an endorsement showing how important a type system is in the development of programming languages. Robin Milner [37, Page 264] emphasis the usefulness of types from a practical point of view:
One of the most helpful concepts in the whole of programming is the notion of type, used to classify the kinds of object which are manipulated. A significant proportion of programming mistakes are detected by an implementation which does type-checking before it runs any program. Types provide a taxonomy which helps people to think and to communicate about programs.

Formally speaking, a type system is composed of a type environment and a set of judgments that are validated using a set of typing rules. A type judgment reflects the fact that, we can use the type environment of the type system and prove that an expression in the corresponding programming language is well-typed (has a type in the defined set of types). A type environment on the other hand, is a mapping from names to their types. We use $E$ to denote a type environment. Take as an example the following type judgment:

$$ E \vdash e : T $$

The above judgment establishes the fact that, it is possible to prove, under the type environment $E$, that the type of expression $e$ is $T$ (we use $T$ to range over types). Normally, the type environment $E$ can be extended with a new name $id$, in the following way:

$$ E, id : T $$

But how are the typing rules implemented? The answer is: by making use of a type-checker. In fact type-checker can be seen as the heart of a type system that implements the typing rules associated with the type system.

To make the concept of the typing rule clear, consider the set of rule shown in Figure 1. It is an example from the book by Pierce [30], and depicts the typing rules for the simply typed lambda calculus.

$$ \frac{x : T \in E}{E \vdash x : T} $$

$$ \frac{E, x : T_1 \vdash t_2 : T_2}{E \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2} $$

$$ \frac{E \vdash t_1 : T_{11} \rightarrow T_{12} \quad E \vdash t_2 : T_{11}}{E \vdash t_1 \; t_2 : T_{12}} $$

Figure 1: Set of type rules for simply typed lambda calculus.
The syntax of the language is as follows:

\[ t ::= \lambda x : T.t \mid t \ t \mid x \]

Note that the type symbol \( T \) that has appeared explicitly in the lambda abstraction. Another alternative would be to use inference techniques in the type-checker in order to infer the type of the arguments in the \( \lambda \)-abstractions. According to Pierce, there are two class of languages regarding this. One class includes the languages in which types appear explicitly in the terms of the language. The members of this class are called *explicitly typed* languages [30]. Type-checker makes use of the explicit type annotations as a guide for type checking the terms of the language. On the other hand, there is a class of languages that use an inference technique (algorithm) deployed by the type-checker to type-check the terms of the language. Members of this class are called *implicitly typed* languages.

Now back to the type rules in Figure 1. The first rule says that to type-check a variable, it must be defined in the typing context \( E \). The second rule is intended for type checking the \( \lambda \)-abstractions. In the premise of the rule, it is assumed that if we extend the type environment \( E \) by variable \( x \) which has type \( T_1 \), and this extended typing context then entails that term \( t_2 \) has type \( T_2 \), we can conclude that the \( \lambda \)-abstraction \( \lambda \) has a function type \( T_1 \to T_2 \) under the type environment \( E \). Finally, the last rule express that if two premises hold, one stating that the abstraction \( t_1 \) has a function type \( T_{11} \to T_{12} \) under \( E \) and the other says that term \( t_2 \) has type \( T_{11} \) under the same type environment, then we can conclude that the result of applying \( t_1 \) on \( t_2 \) has type \( T_{12} \).

During the development of type systems, there has been many improvements and extensions targeting both type systems and types. For example, enriching the type rules of a type system with rules for checking recursive datatypes such as lists or trees is one extension. Another example is the concept of sub-typing (and its corresponding improvements that has been affected the design of modern type systems). Sub-typing is specially very important in the object-oriented languages and in fact plays an essential role in the object-oriented style of programming. Intuitively, a type \( S \) is a subtype of type \( T \), if any term with type \( S \) can be used safely in a context that expects a term of type \( T \).

Every type system must at least possess a property known as *safety* (or sometime called soundness). This intuitively means that a well-typed term does not “go wrong”. For a term to go wrong may have different meanings under different systems. For example the wrong state can be seen as a *stuck*\footnote{For a thorough description on what is a function type refer to the book by Pierce [30, Section 9.1].}
state - a state that while in it, the term is not in its normal form but there is no available transition applicable to it [30].

This safety property is itself composed of two sub-properties. In other words, every type system must satisfy these two properties, namely progress and preservation, which together constitute the soundness (safety) property.

Pierce [30], has defined the progress and preservation properties as follows:

**Progress:** A well typed term is not stuck (either it is a value or it can take a step according to the evaluation rules).

**Preservation:** if a well-typed term takes a step of evaluation, then the resulting term is also well typed (this property is also called the *subject reduction* property).

Later in Chapter 3.3, we discuss the above properties under the context of the psi-calculi type system.

Now that we have described the theory of types and type systems and the application of type systems as a formal method to attest some correctness properties of the programs written in a programming language, we have enough background in order to define the type system of the psi-calculi and its implementation. In the next chapter, we discuss more deeply the typed psi-calculi, their type system and related theoretical and practical issues.
3 Typed psi-calculi

In this section, we present the type system of psi-calculi in detail and investigate its design details and requisites. We look at the different parts of the Psi-calculi Workbench and present the necessary modifications that are required in order to implement the typed psi-calculi. The theories presented in this chapter mostly originate from the paper *Typed ψ-calculi* [17] by Hüttel and the master thesis report by Gutkovas [13].

Psi-calculi is an effort to generalize a family of process calculi that are mostly extensions to the original pi-calculus by Milner, Parrow and Walker [26]. Every member of this family is called an instance. Psi-calculi is a framework for creating process calculi. One just needs to instantiate seven parameters of the framework in an appropriate way and gets the desired process calculus [19, Chapter 3]. Processes communicate with each other based on message passing. A process sends to or receives messages from other processes over channels. A channel is an abstraction of a link between two processes. The result of these interactions, normally, is that the process transits into another process form (computation). This transition might be silent form, meaning that the interaction is hidden from outside world.

Here we present the psi-calculi processes (also called agents) and their informal definitions. The definitions are adapted from [13] with modifications regarding the psi-calculi type system.
Empty agent (0)
Does no action.

Output prefix agent ($\overline{M}.N.P$)
This process performs the output action $\overline{M}N$ and transitions into the process $P$. The process sends the data (called object) $N$ over the channel $M$ (channel $M$ is called subject here).

Input prefix agent ($\overline{M}(x).P$)
After receiving some value ($N$) over the channel $M$, the agent continues as the process $P[x := N]$.

Case agent (case $\varphi_1 : P_1 \parallel \cdots \parallel \varphi_n : P_n$)
If a condition $\varphi_i$ holds, then the process will continue as the subprocess $P_i$ corresponding to that condition. If more than one condition hold, a subprocess is non-deterministically chosen. The process will not perform anything if none of the conditions hold.

Restriction agent ($\nu a : TP$)
The intuition here is that the name $a$ has been made private in the process $P$. Otherwise the restriction agent acts as $P$. The name $a$ is distinct from other names in the process’s environment and it can be transmitted to other processes.

Parallel agent ($P | Q$)
This process form models concurrent execution. These processes can also act independently. Processes $P$ and $Q$ can communicate, issuing a silent transition $\tau$. In that case, the subjects of two processes must be equivalent (same channel), and one of the processes does an input and the other does an output action.

Replication agent ($!P$)
Can be seen as infinitely many copies of $P$ in parallel.

Assertion agent ($L\Psi$)
Interacts with agents in parallel by entailing conditions.

Invocation agent ($A\langle m_1, \ldots, m_n \rangle$)
Process identifier $A$ has a definition as follows:

$$A(x_1 : T_1, \ldots, x_n : T_n) \leftarrow P.$$ 

The intuition here is that the invocation can act as the process $P$ replacing $x_i$ (the formal parameters) with $m_i$ (the actual parameters) for every $i$. It is possible to have multiple definitions for $A$.

The definition above is an informal introduction to the psi-calculus processes. In this definitions, we range over names $N$ (Definition 1) by $a, b, \ldots, z$ and we use $M, N$ to range over terms. Processes are ranged over by $P, Q$ and we use $T$ to range over types. Later we define the formal syntax of the psi-calculus agents.

In typed psi-calculi, the set of types is a nominal datatype, so names can
appear in types. We name the set of types by $\text{Ty}$. Therefore the modified nominal datatypes of the psi-calculi framework is as follows:

**Definition 9** (Typed psi-calculi parameters). A typed psi-calculus requires the four (not necessarily disjoint) nominal data types:

- $T$ the (data) terms, ranged over by $M, N$
- $C$ the conditions, ranged over by $\varphi$
- $A$ the assertions, ranged over by $\Psi$
- $\text{Ty}$ the types, ranged over by $T$

and the four equivariant operators:

- $\leftrightarrow: T \times T \to C$ Channel Equivalence
- $\otimes: A \times A \to A$ Composition
- $\mathbf{1}: A$ Unit
- $\vdash \subseteq A \times C$ Entailment

To determine if two terms represent the same communication channel we use the channel equivalence $\leftrightarrow$ operator. To compose two assertions into one we use the composition $\otimes$ operator. We have the notion of unit assertion (1) which does not introduce any new information to composition relation ($\Psi \otimes 1 \simeq 1 \otimes \Psi \simeq \Psi$). The rationale behind the entailment relation $\vdash$ is that it interprets a condition according to the information given by the assertion. The channel equivalence relation for example, is a condition, therefore communication can be under the control of the environment (assertions here).

Now we give the definition of assertion equivalence relation. Two assertions are said to be equivalent if and only if they both entail the same condition. Or more formally:

**Definition 10** (Assertion equivalence). $\Psi \simeq \Psi'$ iff $\forall \varphi. \; \Psi \vdash \varphi \iff \Psi' \vdash \varphi$

The following definition lists the necessary requirements on the psi-calculi parameters.

**Definition 11** (Requisites on psi-calculi parameters).

- **Channel symmetry:** $\Psi \vdash M \leftrightarrow N \implies \Psi \vdash N \leftrightarrow M$
- **Channel transitivity:** $\Psi \vdash M \leftrightarrow N \land \Psi \vdash N \leftrightarrow L \implies \Psi \vdash M \leftrightarrow L$
- **Composition:** $\Psi \simeq \Psi' \implies \Psi \otimes \Psi'' \simeq \Psi' \otimes \Psi''$
- **Identity:** $\Psi \otimes \mathbf{1} \simeq \Psi$
- **Associativity:** $(\Psi \otimes \Psi') \otimes \Psi'' \simeq \Psi \otimes (\Psi' \otimes \Psi'')$
- **Commutativity:** $\Psi \otimes \Psi' \simeq \Psi' \otimes \Psi$
- **Weakening:** $\Psi \vdash \varphi \implies \Psi \otimes \Psi' \vdash \varphi$
- **Names are terms:** $\mathcal{N} \subseteq T$
In the above definitions, note that the channel equivalence does not require to be reflexive, meaning that we can have data terms that cannot be used as a channel since they are not channel equivalent with any term.

3.1 Typed syntax

Here we give the formal definition of the typed psi-calculi syntax. According to Hüttel [17] the types are introduced into the syntax of psi-calculi in the form which is shown in the following definition:

**Definition 12** (Psi-calculi syntax (Typed version)).

\[
\begin{align*}
0 & \quad \text{Nil} \\
MN.P & \quad \text{Output} \\
M(N).P & \quad \text{Input} \\
\text{case } \varphi_1 : P_1 \mid \cdots \mid \varphi_n : P_n & \quad \text{Case} \\
(\nu a : T)P & \quad \text{Restriction} \\
P | Q & \quad \text{Parallel} \\
!P & \quad \text{Replication} \\
(\Psi) & \quad \text{Assertion}
\end{align*}
\]

In the above definition, the only place that types appear explicitly is in the *Restriction* agent. It has been assumed that in the typed syntax of psi-calculi, the name \( a \) in the restriction agent is annotated with type \( T \). In input agent the object \( N \) binds into the residual \( P \).

According to Hüttel [17], in the nominal datatype of psi-calculi, we use simultaneous term substitution:

\[ X[\tilde{z} := \tilde{M}] \] – terms in \( \tilde{M} \) replace names in \( \tilde{z} \).

For every function symbol \( f \) and term substitution function \( \sigma \) which acts on variable names, we require that the term substitution distributes over function symbols:

\[ f(M_1, \ldots, M_k)\sigma = f(M_1\sigma, \ldots, M_k\sigma) \]

Distributivity is necessary (but not sufficient) for the standard substitution lemma for type judgments to hold if the substitution is well-typed. A substitution is well-type in type environment \( E \), if \( \sigma(x) \) and \( x \) have the same type for any variable \( x \) in the domain of \( \sigma \). This intuitively means that a well-typed substitution function preserves typing after applying it on the name \( x \).

3.2 Typed symbolic operational semantics

In this section we introduce the typed version of the symbolic operational semantics of psi-calculus. We show where type annotations appear in the operational semantics and we give the typed version of the symbolic transition constraints.
In typed psi-calculi, the definition of the frame of a process is different from the untyped version [13, Section 2.3]. Here, the frame of process $P$ not only extracts the assertion information of $P$ ($\Psi_P$), but it also keeps track of the names and their types $E_P$ which are local to $\Psi_P$; therefore the frame of $P$ in typed psi-calculi is as follows:

$$\mathcal{F}(P) = \langle E_P, \Psi_P \rangle$$

Note that $E_P$ doesn’t contain any assertion. According to Hüttel [17], these kind of frames are called qualified frames; because unlike the untyped version of frames, now the names are annotated with their corresponding types. The composition of frames is then defined as follows:

$$\langle E_1, \Psi_1 \rangle \otimes \langle E_2, \Psi_2 \rangle = \langle E_1, E_2, \Psi_1 \otimes \Psi_2 \rangle$$

In the above definition, we require that the following freshness conditions hold

$$\text{dom}(E_1) \neq \text{dom}(E_2) \quad \text{and} \quad \text{dom}(E_1) \neq \Psi_2 \quad \text{and} \quad \text{dom}(E_2) \neq \Psi_1.$$ 

Here by $\text{dom}(E)$ we mean the set of names appearing in $E$. Moreover, assuming that $F$ is a frame, by $(\nu b : T)F$ we mean $\langle b, \Psi_F \rangle$.

The formal definition of the frames of agents in typed psi-calculi is as follows:

**Definition 13** (Frame of an agent (typed)).

$$\mathcal{F}(P \mid Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q)$$

$$\mathcal{F}(\langle \Psi \rangle) = \langle \epsilon, \Psi \rangle$$

$$\mathcal{F}((\nu b : T)P) = (\nu b : T)\mathcal{F}(P)$$

$$\mathcal{F}(P) = 1 \quad \text{otherwise}$$

In the above definition, the only place that type annotations appear is in the frame of the restriction agent; the other processes has the frame as defined in [13, Section 2.3].

Here we need to give the definition of the typed label (action) of the transition rules. The labeled transition rules of the psi-calculi are of the form

$$\Psi \triangleright P \xrightarrow{\alpha} P'$$

where the action $\alpha$ is defined based on the following syntax

$$\alpha ::= \tau \mid MN \mid M(\nu \tilde{b} : T)N.$$ 

Note that in the above definition, the last action clause carries type annotations on the extruded names. Here we have $\text{bn}(M(\nu \tilde{b} : T)N) = \{ \tilde{b} \}$.
and $\text{bn}(\alpha) = \emptyset$ if $\alpha$ is an input or $\tau$ action (By $\text{bn}(\alpha)$ we mean the set of names which are extruded by the action $\alpha$).

To be able to define the symbolic operational semantics of psi-calculi, we need to collect the simultaneously arising constraint which are required to be solved in order to make the transitions valid. We define the appropriate constraints and then decorate the operational semantics with these constraints to achieve the fully operational typed symbolic semantics. To do so, we need to introduce types in transition constraints of operational semantics. In the next definition, we have defined the typed version of the transition constraints.

**Definition 14 (Typed transition constraint).**

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$true$</td>
<td>${ (\sigma, \Psi) : \sigma$ is a subst. sequence $\land \Psi \in A }$</td>
</tr>
<tr>
<td>$false$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$(\nu a : T){\Psi \vdash \varphi}$</td>
<td>${ (\sigma, \Psi') : \exists \tilde{b}. \tilde{b}#\sigma, \Psi' \land ((\tilde{a} \tilde{b}) \cdot \Psi)\sigma \otimes \Psi' \vdash ((\tilde{a} \tilde{b}) \cdot \varphi)\sigma }$</td>
</tr>
<tr>
<td>$C \land C'$</td>
<td>$\text{sol}(C) \cap \text{sol}(C')$</td>
</tr>
</tbody>
</table>

In $(\nu a : T)\{\Psi \vdash \varphi\}$, the names $\tilde{a}$ are binding occurrences into $\Psi$ and $\varphi$. We let $(\nu a)(C \land C')$ mean $(\nu a)C \land (\nu a)C'$, and $(\nu a : T)\text{true}$ means true. Same holds for false; $(\nu a : T)\text{false}$ means false.

Having prepared ourselves with enough requisites, we are now ready to give the definition of the symbolic operational semantics of typed psi-calculus. They are listed in Figure 2.

Now we have all the background theory necessary for integrating types into the Psi-calculi Workbench. The definitions we gave above are mostly applied in the core parts of the Workbench.

### 3.3 Psi-calculi type system

The type system for psi-calculi is an extension to that of the simply typed pi-calculus by Sangiorgi and Walker [33]. The type system of the simply typed pi-calculus is extended in two directions. First, in the psi-calculi type system, messages have been assigned a type; and second, the impact that assertions might have on the typing of a term is taken into account [17].

Every type system is composed of three main parts: The type environment, type judgments and the types. We saw before (in Definition 9) that the set of types in typed psi-calculi framework is a nominal datatype. In the following sections we discuss the issues regarding the type environment and type judgments of the typed psi-calculi framework.
\[
\begin{align*}
\text{In} & \quad \Psi \vdash \overline{M}((x : T)) \cdot P \frac{\frac{y \# \Psi, M, P, x}{\{y : T \}}}{\text{P}} \\
\text{Out} & \quad \Psi \vdash \overline{M} N \cdot P \frac{\frac{\overline{K} N}{\{y : T \}}}{\text{P}} \\
\text{Case} & \quad \Psi \vdash \text{case } \overline{\tau} : \overline{P} \frac{\frac{\alpha}{\{y : T \}}}{\text{P}} \\
\text{Com} & \quad \Psi \vdash P | Q \frac{\frac{\tau}{\text{C}_{\text{com}}}}{(\nu a : T)(P') \mid Q'[x := N])} \\
\text{PAR} & \quad \Psi \vdash P | Q \frac{\frac{\alpha}{\{y : T \}}}{\text{P'} \mid Q'} \\
\text{Scope} & \quad \Psi \vdash (\nu b : T)P \frac{\frac{\alpha}{\{y : T \}}}{(\nu b : T)P'} \\
\text{OPEN} & \quad \Psi \vdash (\nu b : T_1)P \frac{\frac{\alpha}{\{y : T \}}}{\overline{M}((x : T_1) \cdot \overline{P})} \\
\text{INVOCATION} & \quad \Psi \vdash \overline{A}((\nu \overline{x} : \overline{T}, P) \in e) \frac{\frac{\alpha}{\{y : T \}}}{\text{P}} \\
\end{align*}
\]

Figure 2: Typed transition rules of the symbolic operational semantics of psi-calculus. We have omitted the symmetric versions of Com and PAR. In the rule Com, \( C_{\text{com}} \) means \((\nu b \overline{P}) \cdot \overline{Q} \). The other related assumptions which hold for the non-symbolic version [5] also hold here. In rule OPEN we extend the typed sequence \( \overline{a} : \overline{T} \) with \( b : T \).
3.3.1 Type environment

According to Hüttel, the type environment of the psi-calculi type system, records the types of free names in the processes. Typability in the psi-calculi type system also depends on assertions, therefore a type environment \( E \) may also contain assertions \([17]\).

The type environments of psi-calculi type system are defined by the following syntax:

\[
E ::= E, x : T \mid E, \Psi \mid \emptyset
\]

We have defined a SML signature for the type environment of the psi-calculi type system and a SML functor is used to implement the type environment for the psi-calculi instances. The code listings of the signature and the functor can be found in Appendix A.

The next definition is adapted from \([17]\) and defines the a well-formed type environment.

**Definition 15** (Well-formed type environment). A type environment is well-formed \( (E \vdash \diamond) \) if it satisfies the following rules:

\[
\begin{align*}
\text{(EMPTY)} & : \emptyset \vdash \diamond \\
\text{(ASSERT)} & : E \vdash \diamond \quad n(\Psi) \subseteq \text{dom}(E) \quad E, \Psi \vdash \diamond \\
\text{(NAME)} & : E \vdash \diamond \quad x \notin \text{dom}(E) \quad n(T) \subseteq \text{dom}(E) \quad E, x : T \vdash \diamond
\end{align*}
\]

Based on the definition above, an empty type environment is well-typed. In order to extend the type environment \( E \) with the assertion \( \Psi \), we must have the names appearing in \( \Psi \) already in the domain of \( E \). To extend the type environment \( E \) with the typed name \( x : T \), the names appearing in the type \( T \) must already be present in the domain of \( E \) and \( x \) must not be in the domain of \( E \).

The following definitions are from \([17]\) Section 3.1] and deal with extending the type environments.

**Definition 16.** We have that \( E_1 <_T E_2 \) if \( E_1 \vdash \diamond \), \( E_2 \vdash \diamond \), \( E_1 = E_{10}, E_{11}, E_{12}, \ldots, E_{1(k+1)} \) and \( E_2 = E_{10}, u_1 : T_1, E_{11}, \ldots, u_k : T_k, E_{1k}, E_{1(k+1)} \) for some \( u_1 : T_1, \ldots, u_k : T_k \). In this way, \( E <_T E' \) if \( E' \) extends \( E \) with additional type annotations.

**Definition 17.** Let \( E, E' \) be two type environments. We have that \( E \lessdot_A E' \) if \( E' = E, \Psi \) for some assertion \( \Psi \). We let \( <_A \) denote the transitive closure of \( <_0^A \) and let \( < \) denote the least preorder relation containing \( <_T \cup <_A \).
3.3.2 Type judgements and type rules

According to Hüttel, the type system for psi-calculi has judgments for the three nominal data types:

\[ E \vdash M : T, \quad E \vdash \varphi \quad \text{and} \quad E \vdash \Psi \]

A judgment \( E \vdash \mathcal{J} \) is qualified if \( n(\mathcal{J}) \subseteq \text{dom}(E) \)[17]. All the judgement must be qualified with respect to the well-formedness of type environment \( E \).

The following definition holds for the qualified judgments.

**Definition 18.** Suppose \( E \vdash \mathcal{J} \) and \( E' \vdash \mathcal{J}' \) be two qualified judgments. Then we have that \( \mathcal{J} E < \mathcal{J}' E \) if \( \mathcal{J} = \mathcal{J}' \) and \( E < E' \).

We also require the following definition to hold with respect to the qualified type judgments:

What follows is a set of general assumptions that are used in order to establish the usual properties – in particular, the weakening and strengthening properties – of any concrete instance of the type system [17]. These assumptions are used with respect to the type rules of the psi-calculi type system. A trivial assumption is that bindings can be used in the type environment:

\[(\text{VAR}) \quad E \vdash x : T \quad \text{if} \quad E(x) = T\]

Every type rule has zero or more premises and a conclusion that all are qualified judgments. There might also be side conditions present in a typing rule; a side condition is not a qualified type judgment and is a predicate that may depend on judgments in the rule. We denote a dependent side condition by the following symbol: \( \mathcal{X}(\mathcal{J}_E) \).

According to Hützel [17], there are conditions regarding type rules for terms and assertions. In the following list, we give the two conditions which are more related to our work. For a full list, interested reader can refer to the paper on typed psi-calculi [17, Section 3.2].

- Each side condition \( \mathcal{X} \) must be monotone wrt. extensions to the type environment. Let \( \mathcal{J}_E \) be an arbitrary instance of a type rule. Suppose \( \mathcal{X}(\mathcal{J}_E) \) holds and \( \mathcal{J}_E < \mathcal{J}'_E \) and \( \mathcal{J}'_E \) is another rule instance and \( \mathcal{X}(\mathcal{J}'_E) \) also holds. Then we say that \( \mathcal{X} \) is monotone.

- Every side condition must be topical; meaning that removing the unnecessary type annotations will not affect the validity of a side condition. Suppose \( \mathcal{J}_E \) is an arbitrary instance of a rule, and we have that whenever \( \mathcal{J}_E <_T \mathcal{J}'_E \) for some other instances \( \mathcal{J}'_E \) and \( \mathcal{X}(\mathcal{J}'_E) \) holds, then also we have that \( \mathcal{X}(\mathcal{J}_E) \). This way we say that \( \mathcal{X} \) is topical.

The typing rules of the type system of psi-calculi are shown in the Figure 3. Note that we have simplified the In rule and the object is now the
name $x$ rather than the term $N$. This fulfils the requirements that we need in our implementation of the type system of psi-calculi (this way we don’t need to use the pattern matching rule [17, Section 3.2]). These rules are implemented as a type-checker module in the Typed Psi-calculi Workbench and the listing of the code of the type-checker module can be found in Appendix [A].

Note that in the rules In and Out we have a side condition that is a predicate (called compatibility predicate) and must holds when type-checking the Input and Output processes. Here an example can be helpful to express the idea behind this compatibility operator. To implement the simply typed pi-calculus by Sangiorgi and Walker [33] with channel types we let $\leftrightarrow$ be defined as $T_1 \leftrightarrow T_2$ if $T_1 = \text{Ch}(T_2)$ meaning that the side conditions of the type rules In and Out, guarantee a general form of channel safety [17]. In fact, by having this property, we are sure that channels are used to transmit messages of appropriate types. For a complete account on safety in the type system of psi-calculi refer to [17, Section 4.3].

3.4 Process constants

In this section we introduce a typing rule for the Invocation agents and we give a proof for the subject reduction property of the type rule. The proofs and definitions in this section were developed in collaboration with my supervisor, Johannes Borgström.

The Invocation process provides the user a facility to define agent clauses in an environment [13] and then she/he can invoke the agent using the identifier syntax [27]. The symbolic operational semantic of the Invocation process is listed in the Figure 2. Note that in the Invocation rule, we have added the type annotations explicitly in the identifier agent syntax. This is a requirement that is imposed by the limitations of this thesis work (we don’t use any type inference technique).

Before introducing the typing rule for the Invocation process, here we give some definitions which are required by the typing rule and by the proof of its subject reduction property. The first definition defines a well-formed clause environments. This definition is required by the Invocation typing rule.

**Definition 19** (Well-formed clause environment). Let $e \subseteq A \times N \times Ty \times P$ be a clause environment. We say $e$ is well-formed ($e \vdash \Box$) iff for each two clauses $(A_1, \tilde{x}_1, \tilde{T}_1, P_1)$ and $(A_2, \tilde{x}_2, \tilde{T}_2, P_2)$ in $e$ if $A_1 = A_2$ and $\tilde{x}_1 = \rho \cdot \tilde{x}_2$ then $\tilde{T}_1 = \rho \cdot \tilde{T}_2$ and $\tilde{x}_1 : \tilde{T}_1 \vdash P_1$.

In the definition above, $\rho$ is a permutation as defined in Definition [2].

The following definition is required by both the typing rule of the Invocation agent and the proof of its subject reduction property. The tuples we use in
Figure 3: Type rules for the psi-calculi type system. In rule (PAR), we extend the type environment $E$ with the frame of agent $P_1$ before type-checking $P_2$ and vice versa. We require that $\text{dom}(E_{P_1}), \text{dom}(E_{P_2})$ be fresh for $\text{dom}(E)$. In the rules (IN) and (OUT), the type of the subject $M$ and the type of the object $N$ (or $x$ in case of IN rule) must be compatible wrt. a compatibility predicate $\leftrightarrow$. Also note that in the above set of rules, there is no rule present for the Invocation process. In the next section, we go through the details of defining the Invocation typing rule.
this definition are isomorphic (to right) nested dependent pairs. We show the instantiation of the variables in types with actual parameters (substitution of $\tilde{M}$ for $\tilde{x}$) in the above tuple as $\tilde{M}/\tilde{x} : \tilde{T}$.

**Definition 20** (Instantiation of dependently typed arguments of a clause). We define the judgment $\tilde{M}/\tilde{x} : \tilde{T}$ as follows:

\[
\begin{align*}
\text{NonDep} & \quad \tilde{x} \cap n(\tilde{T}) = \emptyset \quad E \vdash \tilde{M} : \tilde{T} \\
& \quad \overline{\quad} \quad \overline{\quad} \quad \tilde{x} \# E \\
\text{Dep} & \quad E \vdash M : T \\
& \quad E \vdash \tilde{N}/\tilde{y} : \tilde{U}[x := M] \\
& \quad \overline{\quad} \quad \overline{\quad} \quad \overline{x \# E, \tilde{y}} \\
\end{align*}
\]

To clarify the definition above, assume the following example which is a tuple with three elements and its type is dependent on the instantiation of the variables $x$ and $y$:

$$(x : T, y : U(x), z : V(x, y))$$

Now suppose that we instantiate the tuple above using values $a, b, c$ for variables $x, y, z$ respectively. Then the resulting tuple $(a : T, b : U(a), c : V(a, b))$ is well-typed if $x, y, z \# E$ and

$$E \vdash a : T \land E \vdash b : U[x := a] \land E \vdash c : V[x := a, y := b].$$

Now we turn to the typing rule for the **Invocation** agent. We define it as follows:

\[
\begin{align*}
\text{Inv} & \quad e \vdash \bullet \\
& \quad (A, \tilde{x}, \tilde{T}, P) \in e \\
& \quad E \vdash \tilde{M}/\tilde{y} : (\tilde{x} \tilde{y}) \cdot \tilde{T} \\
& \quad \overline{\quad} \quad \overline{\quad} \quad \tilde{y} \# E \\
\end{align*}
\]

The rule Inv has three premises. The first premise states that the clause environment $e$ that the must be well-formed. Of course the agent identifier must be defined in the clause environment $e$ (indicated by the second premise). The last premise is based on Definition 20. Note that in the last premise of Inv rule we need to permute the names $\tilde{x}$ which might already be in the environment $E$ with fresh $\tilde{y}$, in order to apply rules Dep or NonDep above.

In the proof of subject reduction property, we use the following standard lemmas which are adapted from [17, Section 4.1].

**Lemma 21** (Standard lemmas). The following propositions hold:

- **Weakening**: If $E \vdash J$ and $E, E' \vdash \bullet$ and $\text{dom}(E') \# J$ then $E, E' \vdash J$.
- **Strengthening**: If $E, x : T \vdash J$ and $x \notin n(J)$ then $E \vdash J$. 

27
• **Substitution**: If \( E, x : T, E' \vdash J \) and \( E \vdash M : T \) then \( E, E'[x := M] \vdash J[x := M] \).

**Proof.** We here consider only \( J = \tilde{M}/\tilde{x} : \tilde{T} \), since the proof of this lemma for the other judgments follow from \([17, \text{Section 4.1}]\). By Definition \([20]\) we have two cases for the inductive derivation of \( J \):

**NONDEP** Here the conclusion follows directly by the freshness of \( \tilde{x} \) and the induction hypothesis: that the statement of the lemma holds for each of the term typing judgments \( E \vdash M_i : T_i \).

**DEP** There are three cases:

**Weakening** Here we have \( E, E' \vdash \circ \) and \( \text{dom}(E') \# x, \tilde{y} \).

\[
\text{DEP} \quad \frac{E \vdash M : T \quad E \vdash \tilde{N}/\tilde{y} : \tilde{U}[x := M]}{E \vdash M, \tilde{N}/x,\tilde{y} : T, \tilde{U}} \quad \text{by Dep.}
\]

Since \( E \vdash \tilde{N}/\tilde{y} : \tilde{U}[x := M] \) and \( x \# \text{dom}(E) \) and also \( x \# T \) we have that \( E, E' \vdash \tilde{N}/\tilde{y} : \tilde{U}[x := M] \) and \( E, E' \vdash M : T \). Since \( x \# E' \) we finally get that \( E, E' \vdash M, \tilde{N}/x,\tilde{y} : T, \tilde{U} \) by **DEP**.

**Strengthening** Here we have

\[
\text{DEP} \quad \frac{E, y : U, E' \vdash M : T \quad E, y : U \vdash \tilde{N}/\tilde{y} : \tilde{U}[x := M]}{E, y : Y \vdash M, \tilde{N}/x,\tilde{y} : T, \tilde{U}} \quad \text{by Dep.}
\]

Since \( y \# M, T \) we have \( E \vdash M : T \) by induction. Since \( y \# \tilde{N}/x,\tilde{y} : \tilde{U} \) we also have \( E \vdash \tilde{N}/x,\tilde{y} : \tilde{U} \) by induction. We conclude \( E \vdash M, \tilde{N}/x,\tilde{y} : T, \tilde{U} \) by **DEP**.

**Substitution** : Here we have \( E \vdash N' : U \) and

\[
\text{DEP} \quad \frac{E, z : U, E' \vdash M : T \quad E, z : U, E' \vdash \tilde{N}/\tilde{y} : \tilde{U}[x := M]}{E, z : U, E' \vdash M, \tilde{N}/x,\tilde{y} : T, \tilde{U}} \quad \text{by Dep.}
\]

By induction \( E, E'[z := N'] \vdash (M : T)[z := N'] \) and \( E, E'[z := N'] \vdash (\tilde{N}/\tilde{y} : \tilde{U}[x := M])[z := N'] \). By **DEP** we finally get \( E, E'[z := N'] \vdash (M, \tilde{N}/x,\tilde{y} : T, \tilde{U})[z := N'] \).

\[\square\]

We also use the following two lemmas in the proof of subject reduction property.
Lemma 22.
If \( E \vdash \tilde{M}/\tilde{x} : \tilde{T} \) then \( E \vdash \tilde{M} : \tilde{T}[\tilde{x} := \tilde{M}] \).

Proof. Assume \( E \vdash \tilde{M}/\tilde{x} : \tilde{T} \). Based on Definition 20 we have two inductive cases:

1. \[ \tilde{x} \cap \pi_{\tilde{T}}(\tilde{T}) = \emptyset \quad E \vdash \tilde{M} : \tilde{T} \]
   \[ E \vdash \tilde{M}/\tilde{x} : \tilde{T} \]
   Since \( \tilde{x} \cap \pi_{\tilde{T}}(\tilde{T}) = \emptyset \) therefore we have that \( \tilde{T}[\tilde{x} := \tilde{M}] = \tilde{T} \) and the result holds.

2. \[ E \vdash \tilde{M} : T \quad E \vdash \tilde{N}/\tilde{y} : \tilde{U}[x := \tilde{M}] \quad x \# E, \tilde{y} \]
   \[ E \vdash \tilde{M}, \tilde{N}/\tilde{x}, \tilde{y} : \tilde{T}, \tilde{U} \]
   Since \( E \vdash \tilde{N} : (\tilde{U}[x := \tilde{M}])[\tilde{y} := \tilde{N}] \) we have that
   \[ E \vdash \tilde{N} : \tilde{U}[x\tilde{y} := M\tilde{N}] \].
   Which means that
   \[ E \vdash \tilde{M}, \tilde{N} : (T, \tilde{U})[x, \tilde{y} := M, \tilde{N}] \] and the result holds.

\[ \square \]

Lemma 23.
If \( E, \tilde{x} : \tilde{T} \vdash J \) and \( E \vdash \tilde{M} : \tilde{T}[\tilde{x} := \tilde{M}] \) then \( E \vdash J[\tilde{x} := \tilde{M}] \).

Proof. By induction on the length of \( \tilde{x} \). The base case \( |\tilde{x}| = 0 \) is trivial.
Now we assume the case for \( |\tilde{x}| = n \) and prove the statement for \( |\tilde{x}| = n + 1 \), i.e. \( \tilde{y}\tilde{x} : U\tilde{T} \). We must show that if \( E, y : U, \tilde{x} : \tilde{T} \vdash J \) and \( E \vdash N : U \) and \( E \vdash \tilde{M} : \tilde{T}[\tilde{y}\tilde{x} := N\tilde{M}] \) then we have that \( E \vdash J[y\tilde{x} := N\tilde{M}] \).

Since \( y, \tilde{x} \# N, \tilde{M} \) we have \( [y\tilde{x} := N\tilde{M}] = [y := N][\tilde{x} := \tilde{M}] \), so by weakening \( E, y : U \vdash \tilde{M} : (\tilde{T}[y := N])[\tilde{x} := \tilde{M}] \). By induction \( E, y : U \vdash (J[y := N])[\tilde{x} := \tilde{M}] \), so we can apply substitution lemma and we have that
\[ E \vdash J[y\tilde{x} := N\tilde{M}] \]
and the lemma holds.

\[ \square \]

Now we give the definition of subject reduction property of the psi-calculi type system and its proof for the Inv type rule. According to Hüttel [17, Section 4.2], the subject reduction property of the psi-calculi type system is as follows:
**Theorem 24** (Subject reduction). If $E \vdash \Psi$ and $E \vdash_e P$ and $\Psi \triangleright_e P \xrightarrow{C} P'$ then either $\alpha = \tau$ and $E \vdash_e P'$, or for all $E'$ such that $E, E' \vdash_e \alpha$ then $E, E' \vdash_e P'$.

**Proof.** By induction on the derivation of 

$$\Psi \triangleright_e P \xrightarrow{C} P'$$

Cases In, Out, Case, Com, Par, Scope, Open, Rep follow from the proofs given by Hüttel in [17].

**Case Invocation:** Inverting the transition rule (Figure 2), we have that $(A, \bar{x}, T, P) \in e$ and $\Psi \triangleright_e P[\bar{x} := M] \xrightarrow{C} P'$. From $E \vdash_e A(M)$ we get that $E \vdash \tilde{M} / \tilde{y} : (\bar{x} \tilde{y}) \cdot \tilde{T}$, where we assume that $\tilde{y} \# E, \tilde{M}$, and let the permutation $\rho = (\bar{x} \tilde{y})$. By Lemma 22, we have

$$E \vdash \tildem : (\rho \cdot \tilde{T})[\tilde{y} := \tildem].$$

(1)

Since $e \vdash \phi$, we get $\tilde{x} : \tilde{T} \vdash_e P$. By equivariance (Definition 6), $\rho \cdot (\tilde{x} : \tilde{T} \vdash_e P) = \tilde{y} : \rho \cdot \tilde{T} \vdash_e \rho \cdot P$ also holds. By the weakening lemma we have (since $\tilde{y} \# E$):

$$E, \tilde{y} : \rho \cdot \tilde{T} \vdash_e \rho \cdot P$$

(2)

We apply Lemma 23 to (1) and (2), yielding $E \vdash_e \rho \cdot P[\tilde{x} := \tildem]$. Then, by equivariance $E \vdash_e P[\bar{x} := \tildem]$. The result follows by the induction hypothesis for the transition of $P[\bar{x} := \tildem]$.

In this section we presented a rule for typing the Invocation agents and gave a proof for the corresponding subject reduction property. Note that the identifier agent may contain recursive calls to itself thus providing recursion in the psi-calculi model of computation. Our typing rule prove that the recursive agents remain well-formed after invoking the agent with actual parameters (the subject reduction property).

### 3.5 Implementations

In this section we describe the details regarding the implementation of the type system of psi-calculi in Psi workbench framework. Terms such as signature, structure and functor which are used in this section are adapted from [35]. We start by explaining the implementation of the type environment (it is according to Definition 3.3.1). Then we discuss the implementation details of the type-checker and we show how we have implemented it in the tool. Other modifications which we have done in the tool regarding the type system implementation are also described in this section.
To design the interface of the different parts of the type system of psi-calkuli, we used the SML module system and this helped a lot to keep the design as generic as possible. Using signatures also facilitates the process of separating the concepts of design from the implementation issues.

Now we turn to explain the details regarding the implementation of the type environment. In every instance of the typed psi-calkuli, we need to instantiate a SML structure as the type environment of that instance.

In order to instantiate the type environment in every specific instance, we use a functor (called TypeEnv) which takes a structure and implements the type environment as a structure. The signature of the structure that is passed to the TypeEnv functor is called ORD_KEY. The ORD_KEY signature and the TypeEnv are shown in the following listing:

Listing 1: Interface to the type environment of the psi-calkuli type system

```sml
signature ORD_KEY =
  sig
    structure Assr: NOMINAL
    structure Type: NOMINAL
    sharing type Assr.atom = Type.atom
    val compose : Assr.data → Assr.data → Assr.data
    val unitAssr : Assr.data
  end
functor TypeEnv (Key : ORD_KEY) : TYPE_ENVIRONMENT
```

The user must provide this structure with appropriate parameters when calling the TypeEnv functor. Two structures must be passed to the TypeEnv functor. One is the Assr structure which corresponds to assertion nominal datatype and the other one is Type structure which corresponds to the nominal datatype of type. These two structures are used to check the well-formedness of the type environment (according to Definition 15) when inserting new name and type pairs or new assertions into the current type environment. We also need to pass the assertion composition operator (⊗) as well as the unit assertion (1) when calling the TypeEnv functor (these values vary in every specific instance accordingly). Unit assertion is used to represent the status of the type environment when there is no assertion in it. Composition operator is used when we insert new assertions into the current environment (we compose the new assertion with current assertion which is already in the type environment). The complete SML code of type environment and other signatures we discussed about in the above text as well as the corresponding functor can be found at Appendix A. To implement the type environment of the type system we use the concept of association lists which facilitates a lot the implementation of required operations such as inserting of new name and type pairs or new assertions into the current type environment and also makes it easy to search for the type of a given
name in the current type environment etc.

One of the important parts of this project is to implement the type-checker of the psi-calculi type system. Here we define a SML signature as the interface to the type-checker in the framework. This signature has two structures; one of them is a structure of the frame (Definition 13) signature and the other one is a structure that implements the clause environment (the definition of clause environment can be found at Section 2.5)). Both structures are used by the type-checker declaration. The signature has also a function declaration called `typeCheck` that declares the type-checker function arguments and its return value. The type-checker function takes a type environment (through the frame structure), an agent clause environment and a psi agent and type-checks the agent according to the typing rules of the psi-calculi type system (these rules are listed in Definition 3). We also defined an exception called `Type` inside the signature of the type-checker to handle the cases which there are typing errors in an agent. If `typeCheck` function can not type-check an agent, the `Type` exception will raise with the appropriate error message. The `TYPE_CHECKER` signature is as follows:

Listing 2: Signature of the type-checker module

```ml
signature TYPE_CHECKER =
  sig
    structure Frm  : FRAME
    structure ClEnv : PSI_CLAUSE_ENVIRONMENT
    exception Type of string
    val typeCheck : Frm.Psi.Inst.Env.env → ClEnv.env → Frm.Psi.psi → unit
  end;
```

A functor is used to implement the type-checker signature as a structure. The full code listing of this functor can be found in Appendix A.

Other important part of the implementation of the psi workbench that we have modified is the code related to the command interpreter. We have extended the set of commands provided by the command interpreter with some new commands in order to make the command interpreter able to support the type requirements of the typed psi workbench. To implement these commands we modified the command interpreter code accordingly. The list of these commands and their full descriptions can be found at section 4.1.2.

Finally, to be able to parse the types in the agents and also pretty print them, we made changes in the `psi-parsing.ML` and `pp.psi.ML` files based on the requirements of the parsing and pretty printing of types. Interested reader can refer to [20] for the full listing of the command interpreter file and other files related to the implementation of the typed psi workbench.
4 Typed Psi-calculi Workbench

In this section we give a brief introduction to the Psi-calculus Workbench. Our focus is on the modified parts of the tool. We explain what changes we have made in different modules of the workbench in order to introduce the concept of types in the Workbench. For a more thorough description about the tool in general, one can refer to the master thesis report by Gutkovas [13]. We also give sample implementations to show the reader how one can instantiate a desired process calculus, providing the appropriate parameters for the workbench.

4.1 Tool

4.1.1 Typed syntax

The Psi-calculi Workbench has three levels of syntactical categories (this is inspired by the Isabelle syntax [28]). According to Gutkovas, these three categories are as follows:

- **Command interpreter.** Handled by the command interpreter parser.
- **Psi-calculus agents.** Handled by the Psi-calculus agent parser, which is a part of the command interpreter parser.
- **Instance.** Handled by the user implemented parsers.

All the three categories above have been modified in order to fulfill the requirements of the typed psi-calculi workbench. In this section we only focus on the modifications that we have made in the agents syntax. We have summarized the typed syntax of the psi-calculi agents in Table 1. The input must be given in the ASCII form. In Table 1 terms are ranged over by $M, N$. Conditions are ranged over by $\phi$, assertions ranged by $\Psi$ and names are ranged over by $a, x$; types are ranged over by $T$. The untyped psi-calculi agents syntax can be found in [13, Table 1].

![Table 1](image)

The syntactical forms in Table 1 are listed in a descending precedence order.

For the complete formal grammar of the Typed Psi-calculi Workbench refer to the Appendix [13]. The grammar listed in the Appendix [13] is the typed version of the original grammar of the Psi-calculi Workbench [13, Appendix D].

In the original typed syntax of psi-calculi agents, the types are not present in the Input agents (both forms) and Invocation agent. But we have implemented the agent parser in a way that types must be given explicitly while parsing those agent in command interpreter. This way we do not need to deploy any type inference technique to infer the type of the associated terms (which is beyond the scope of this project).
4.1.2 Extended command interpreter

To introduce the concept of types in the Psi-calculi Workbench, one step is to extend the command interpreter of the tool to give it the ability to manipulate the types and type environments of each instance. Currently the command interpreter of the workbench is equipped with some commands that are used to manipulate and analyze the agents. For a complete list of the commands supported by the workbench command interpreter one can refer to [13, Section 3.1.2]. Command that we use from untyped Psi-calculi Workbench’s set of commands are as follows:

- **sstep** $P$ enters the *strong* symbolic execution simulator for the agent $P$. Simulator supports the following subcommands, which must be separated by a *newline*.
  
  - $N$ where $N$ is the number of a transition derived by the simulator. Upon entering this number the simulator chooses the derivative with this number and computes new transitions from that derivative.
  - `b` backtracks to the previous derivative.
  - `q` quits the simulator.

- **env** prints the current process clause environment.

- **drop** $A$ removes all process clauses associated with the constant $A$, from the environment.

- **def** \{ $A(x_1:T_1,x_2:T_2,...) <= P$ ; $B(x_1:T_1,x_2:T_2,...) <= Q$ ; ... \}.

---

<table>
<thead>
<tr>
<th>Form</th>
<th>Notation</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>$\exists N.P$</td>
<td>'M&lt;N&gt;.P</td>
</tr>
<tr>
<td>AbbrInput</td>
<td>$\text{M}(x).P$</td>
<td>M(x : T).P</td>
</tr>
<tr>
<td>Input</td>
<td>$\exists (\lambda \exists x)N.P$</td>
<td>M(\x1 : T1, ... , (xn : Tn))N.P</td>
</tr>
<tr>
<td>Restriction</td>
<td>$\nu a : T P$</td>
<td>(new (a : T))P</td>
</tr>
<tr>
<td>Replication</td>
<td>!P</td>
<td>!P</td>
</tr>
<tr>
<td>Assertion</td>
<td>$</td>
<td>\Psi</td>
</tr>
<tr>
<td>Invocation</td>
<td>$\text{A}(\exists N)$</td>
<td>A&lt;M1, ..., Mn&gt;</td>
</tr>
<tr>
<td>Parallel</td>
<td>$P</td>
<td>Q$</td>
</tr>
<tr>
<td>Case</td>
<td>$\phi_1 : P_1$</td>
<td>case phi1 : P1</td>
</tr>
<tr>
<td></td>
<td>$\ldots$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\phi_n : P_n$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Typed agent forms handled by the parser.
Inserts process clauses into the environment. Process clauses are separated by a semicolon ‘;’. Any number of clauses can be given. Newlines and other whitespace are insignificant. Clauses can be given the same name, introducing non-determinism, and the clauses may be mutually recursive. Note the explicitly given type annotations in the definition of process clauses. The command \texttt{def \{A(x_1:T_1,x_2:T_2,...)<= P;\}} can be abbreviated as \texttt{A(x_1:T_1,x_2:T_2,...)<= P;}: 

Note that in the command interpreter, every command must be separated by a semicolon ‘;’. If a command is not recognized by the interpreter, then the command interpreter will treat the input as an agent and will pass it to the \texttt{agent} command.

We extend the set of commands of the Workbench by the following commands.

- \texttt{typecheck \ P}
  Passed the agent \( P \) to the type-checker module. If the agent is not well-typed, outputs the appropriate error messages.

- \texttt{deftype \{ \ a_1 : T_1 \ ; \ldots \ ; \ a_n : T_n \ \}}
  Inserts \((\text{name},\text{type})\) pairs into the type environment. Each pair is separated by a semicolon ‘;’. Any number of pairs of type definitions can be given. Whitespace and newlines are not important. If same names appear in the definition of types, only the first one will be inserted into the type environment.

- \texttt{tyenv}
  Prints the current type environment.

- \texttt{dropty \ a}
  Removes the name \( a \) and its corresponding type from the current type environment.

To load the command interpreter of workbench and then load the psi-calculus instance, one shall use a running Poly/ML interpreter. As an example the typed pi-calculus instance can be loaded by the following commands:

\begin{verbatim}
use "ROOT.ML";
use "pi2.ML";
Pi.start();
\end{verbatim}
In the above command, \texttt{ROOT.ML} is the main workbench SML file that contains all the requisites for implementing a typed psi-calculus instance using the tool. The file \texttt{pi2.ML} includes the definition of the typed pi-calculus instance (for the complete description see the section \ref{subsec:pi-calculus-instance}). The function \texttt{pi.start} starts the workbench command interpreter.

\section{Implementation of Psi-calculus instances} \label{sec:implementation}

In this section we give two examples of implementing typed psi-calculus instances in the workbench. We show how to instantiate a typed psi-calculus instance by providing the appropriate parameters in the typed version of the psi workbench. The first instance is the typed pi-calculus instance where the terms are simple names; the second one is the typed distributed pi-calculus instance when the terms are structural terms composed of two names \cite[Section 5.2]{17}.

The typed pi-calculus instance theory is adapted from \cite{33}. For the implementation of the typed distributed pi-calculus instance we used theories form both \cite{32} and \cite{17}. We discuss the details regarding the implementation of the psi-calculi parameters in each instance in a way that satisfies the requirements of the instance. We also explain the implementation details of the required functions and operations for each instance specific psi-calculi datatype.

We start by the pi-calculus instance, and its implementation details. We put more efforts to cook the first example as it is more straightforward and easy to establish and investigate comparing to the other example.

\subsection{Pi-calculus instance} \label{subsec:pi-calculus-instance}

The first instance we are going to describe (and show the implementation of it in the tool) is the typed pi-calculus instance \cite[Part 3]{33}. It does not include sophisticated features such as non trivial assertions, or terms with binders. Consequently, we choose the typed pi-calculus instance as the first example of instance implementation and we are save from concerning about more elaborate details.

The implementation is done based on the following steps:

\begin{itemize}
  \item First we define the appropriate parameters for the instance (with respect to types). This includes the nominal datatype representing types, equivariant operators, substitution functions and the function deciding alpha equality \footnote{Due to the fact that the operators of the typed pi-calculus instance are exactly the same as the untyped version, we will omit those operators implementation here and refer the interested reader to the master thesis report by Gutkovas \cite[Section 3.2.1]{13}}.
\end{itemize}
The second step is to define the requirements associated with the type system:

1. Requirements for the type environment $E$.
2. Translation of types from BNF to SML datatypes.
3. Implementing the instance-specific functions required by the type-checker module.

These requirements are declared in the $\texttt{PSI\_INSTANCE}$ signature.

The final step is to implement the functions responsible for parsing and pretty-printing of types.

**Instance definition**

The first step is to define the nominal datatype corresponding to the types in the simply typed pi-calculus (we know that types are also nominal datatype under psi-calculus terminology). Here we give the definition of the nominal datatypes for typed pi-calculus instance.

$$
\begin{align*}
    T & \overset{\text{def}}{=} \mathcal{N} \\
    C & \overset{\text{def}}{=} \{ a = b : a, b \in T \} \cup \{ \top \} \quad \text{where} \quad \top \not\in T \\
    A & \overset{\text{def}}{=} \{ 1 \} \\
    Ty & \overset{\text{def}}{=} \text{Base} \cup \{ \text{Ch}(T) : T \in Ty \}
\end{align*}
$$

In the list above, the set of types ($Ty$) is based on the definition of types in simply typed pi-calculus by Sangiorgi and Walker [33]. The other three nominal datatypes definitions are based on [13, Section 3.2].

The translation of types into the SML datatypes is as follows:

```sml
datatype ty = Base | Chan of ty
```

There is a set of base types, which for the sake of simplicity we use the single $\texttt{Base}$ constructor to represent them (base types can be for example integer types and/or boolean types). There is also a set of channel types in the simply typed pi-calculus instance ($\texttt{Chan}(t)$) which represent the channels that are able to carry terms of type $t$.

Next we show the implementation of the equivariant functions for the types. The interested reader can refer to the master thesis by Gutkovas [13, Section 3.2.1] for the definitions of these functions for the other nominal
datatypes in the pi-calculus instance. The first one is the support function. We represent the support with a set of names. The function `supportTy` computes the support for the types. According to the fact that names don’t appear in the types in pi-calculus instance, the support function simply returns an empty list. The next function to implement is the swap function for types. Its implementation is also straightforward since names don’t appear in types. We also require to implement the substitution function on types. A substitution function has two arguments. The first one is a substitution sequence `\sigma` (which is a list of name and term pairs) and the second one is an instance of a nominal datatype. We name the substitution function for the types as `substTy`. Note that here again, based the fact that names are not present in the types, the implementation of the substitution function is quite straightforward in the pi-calculus instance (we simply omit the substitution sequence and return the type itself). The following listing shows the implementation of the three functions we described above:

```
fun supportTy _ = []
fun swapTy _ t = t
fun substTy _ t = t
```

The last function is the equality function. The instance requires to provide a function which decides \(\alpha\)-equality for types with binders. Since types—which are nominal—don’t feature binders, the \(\alpha\)-equality in this case is just syntactic equality. The implementation of this function is as follows:

```
fun eqTy _ (a,b) = a = b
```

In the function above, the first argument which has been ignored here, is a function that when called, decides \(\alpha\)-equality for datatypes with binders. For a more thorough description on this, and an example of how this function will be implemented in case of a datatype with binders see [13, Section 3.2.1].

We are now done with the implementation of the nominal datatypes and their corresponding equivariant functions. The next step is to define the type environment for the pi-calculus instance.

### Implementation of type environment and generic functions

To implement the type environment for the type system of pi-calculus instances, we have defined a SML functor that implements a structure corresponding to the `TYPE_ENVIRONMENT` signature. This functor is named `TypeEnv` and takes as argument a signature named `ORD_KEY` that defines the keys and
values that are going to be implemented in the current type environment. These signatures and the functor are listed in the Appendix A.

In the pi-calculus instance the Env structure is implemented as follows:

```
Listing 7: Type environment for Pi-calculus instance

structure Env = TypeEnv( struct
    structure Assr = Assr
    structure Type = Type
    val unitAssr = Unit
    fun compose _ _ = Unit
    end)
```

The user is supposed to define the appropriate parameters for the TypeEnv functor, in order to achieve the desired functionalities with respect to the instance specific type environment.

The type environment for pi-calculus instance keeps track of (key,value) pairs. We pass two structures Assr and Type which are used by the functor to control the well-formedness of the type environment as well as two provide the required key value pairs when inserting new name and its type or new assertions into the current type environment. The compose function implements the assertion composition operator and unitAssr value is required when the type environment is going to be initialized by the functor.

To finish the definitions in this section, there are few functions implementations remaining which all are instance specific functions used by the type-checker module to type check the terms of the instance. The declaration of these functions can be found in the PSI_INSTANCE signature in the psi.ml file. In addition to these type-checker related functions, there are two other functions declared in the PSI_INSTANCE signature that are responsible for the implementation of the side condition of the typing rules of In and Out.

The first function is named checkT and is supposed to search for the term \( t \) in the environment \( e \) and if successful, it will return the type of the term \( t \), otherwise a TypeErr exception will raise. The implementation of this function in the pi-instance is as follows:

```
Listing 8: checkT implementation for Pi-calculus instance

fun checkT e t =
    case Env.find e t of
    SOME ty ⇒ ty
    | NONE ⇒ raise TypeErr
        ("The term " ^ t ^ " is not defined in the type environment")
```

39
This function is used by other functions inside the TypeChecker functor; one example use is to type check the object and subject terms of an Input agent. This function is an instance specific one, meaning that the implementation may differ from one instance to the other. The next function we are going to see is the checkC function. This function is used by the type-checker module while type checking the Case agents. The implementation of this function in the pi-calculus instance is as follows:

Listing 9: checkC implementation for pi-calculus instance

```ml
fun checkC e c =
  case c of
  | Eq (a, b) ⇒
    let
      val t1 = checkT e a
      val t2 = checkT e b
    in
      if (t1 = t2) then ()
      else raise TypeErr
         ("condition type error")
    end
  | T ⇒ ()
```

In the above function, in the case of the condition of Eq(a,b), it first uses the checkT function to type-check the terms that constitute the equality condition and then it checks to see if these two term are of the same type. If ‘yes’ then this means the overall condition is well typed; otherwise the exception will raise. In the pi-calculus instance we have another condition (the always entailed condition T). In the case of T, we consider it as a well-typed condition (the checkC function simply returns a Unit value).

The last function is the one that type-checks the assertions. Based on the fact that in the pi-calculus instance, assertions are only the unit assertion, there is no need to care about this and we consider the assertions of pi-calculus instance well-typed. The checkA function implementation is as follows:

Listing 10: checkA implementation for pi-calculus instance

```ml
fun checkA e a = ()
```

We have seen in Figure 3 that in the typing rules of IN and OUT, we have side conditions regarding the compatibility predicate ⇒. We have implemented these side conditions as two instance specific functions named compatIn for the IN rule and compatOut associated with the OUT rule.

In simply typed pi-calculus, the compatibility operator is defined as $T_1 \Rightarrow T_2$ if we have that $T_1 = \text{Ch}(T_2)$. The implementation of these
functions is as follows:

```haskell
fun compatIn _ Us Uo = (Us = Chan(Uo))
fun compatOut _ Ts To = (Ts = Chan(To))
```

Table 2: The ASCII representation of the nominal datatypes corresponding to the typed Pi-calculus instance.

<table>
<thead>
<tr>
<th>Notation</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a \leftrightarrow b</td>
<td>a = b</td>
</tr>
<tr>
<td>⊤</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Base</td>
<td>Base</td>
</tr>
<tr>
<td>Chan(t)</td>
<td>Ch(t)</td>
</tr>
</tbody>
</table>

Note that in both functions, the same criteria holds; meaning that two types are compatible if the subject is a channel able to carry the corresponding object.

Now we have all necessary requisites for the pi-calculus instance implementation and the `PiInstanceRequirements` structure is now complete. The next section is about implementing the parsers and printers for the typed pi-calculus instance.

### Implementation of printers and parsers

The final part of defining an instance and make it able to run in the tool, is to define the appropriate functions for parsing and printing the nominal datatypes.

Before we go on, check the table that summarizes the ASCII syntax of the nominal datatypes of the typed pi-calculus instance. Names will be represented in an alphanumeric form. The equality conditions are formed by =, the always entailed condition is written ⊤ and the unit assertion is shown by 1. The types are written by `Base` for the basic types and `Ch(t)` for the channel types.

We now explain the implementation of the printer function for printing the types. The functions for printing other nominal datatypes are fully explained in the thesis report by Gutkovas [13, Section 3.2.1].

According to the fact that names are represented by strings in the pi-calculus instance, therefore the printing functions are almost trivial. The printing function for types is as follows:
Listing 12: Printing the types in the pi-calculus instance

```haskell
fun printTy (Base) = "Base"
  | printTy (Chan t) = "Ch(" ^ printTy t ^ ")"
```

Now we turn to explain the parsing of the types for the pi-calculus instance. The parsing of the nominal datatypes in the workbench is done using the combinator library that is provided by the tool.

Before we start, we need to construct a parser structure using the `Parser` functor which will help us by providing the parsing combinators on string streams. This is done as follows:

Listing 13: Implementation of parser structure

```haskell
structure Parser = Parser(StringStream)
```

The next step is to use the `PsiParserBase` functor and pass it the above structure (Parser structure) and implement a structure that will provide us with the psi-calculus basic lexical parser combinators such as whitespace, identifier, etc.

Listing 14: Implementation of parser structure

```haskell
structure Lex = PsiParserBase(Parser)
```

For more details on the parsing combinators the interested reader can refer to Gutkovas [13, Page 43].

The `Lex` structure provides us with the lexical parser combinators. These lexical combinators are associated with the lexical rules of the Psi-calculi Workbench (Section 4.1.1).

Now we have the necessary prerequisites and we can define the parsers to parse the nominal datatype of types.

The syntax of the types of the typed pi-calculus instance can be given using the following grammar.

\[
\begin{align*}
\langle \text{channel-type} \rangle & ::= \text{`Ch(} \langle \text{pi-types} \rangle \text{`)} \\
\langle \text{base-type} \rangle & ::= \text{`Base'} \\
\langle \text{pi-types} \rangle & ::= \langle \text{channel-type} \rangle | \langle \text{base-type} \rangle
\end{align*}
\]

To parse the types based on the above syntax, we have implemented a function using the parser combinator as follows:

Listing 15: The parser function for types

```
42
```
fun typ () =
    (Lex.stok "Base" >>= Parser.return Base)
  </Parser.choice/>
  (Lex.stok "Ch" >>= Lex.stok "(" >>=
    (Parser.delayed typ) >>=
    (fn t ⇒ Lex.stok ")" >>= Parser.return (Chan t)))

There are some points to mention about the typ () function. We have used stok combinator to parse the syntactical elements. The choice combinator is used to make the function able to parse both type constructors in a deterministic fashion; meaning that it first tries to parse a Base type and if not successful, it will try to parse a Channel type. The Parser.return function returns a Sml value which corresponds to the result of the parser combinator. The Parser.delayed function is used to call the typ () function recursively and parse the nested types.

For the complete explanation of the parsing the other nominal datatypes (terms, conditions and assertions) and their related issues, one can refer to the report by Gutkovas [13, Section 3.2.1].

The final form of the required parsing function for types is as follows:

Listing 16: parseTy function required by the Psi-parser

fun parseTy s = parseResult (typ ()) s

The parseResult function is used to fill the gap between the requirements imposed by the parsing functions of nominal datatypes and the requirements of the parsing function from combinator library [13, Section 3.2.1].

The parseTy function is declared in the PSI_PARSER_REQ signature under the psi−parsing.ML file.

This is the end of the implementation of typed pi-calculus instance in Psi-calculi Workbench.

4.2.2 Sample run sessions for pi-calculus instance

In this section, we show sample sessions of interaction with the command interpreter using the typed pi-calculus instance. To load the command interpreter, we follow the steps as described at the end of Section 4.1.2. We use the style established by Gutkovas in his master thesis report [13], to typeset the command interpreter’s prompt:

\[
\text{psi}>
\]

the input text will be as

\[
\text{command_name argument1 argument2 ... ;}
\]

and the output of the command interpreter as
Our first example is a parallel agent. Suppose we have the following agent as a running example and the current type environment is empty:

\[ \text{a} (x : \text{Base}). \overline{\text{hello}} x \mid \pi \text{world} \]

Passing the above agent to the type-checker module, must raise a type error. Although the name \( x \) in the right hand side agent is explicitly typed, but the free name \( a \) is not defined in the current type environment (we omit the right hand side agent). Therefore if we try the above agent in the loaded command interpreter, using \texttt{typecheck} or \texttt{sstep} commands, we expect the type-checker to throw an exception, causing the command interpreter to output an error message. Now let us try the above agent in a loaded command interpreter to see what happens.

\begin{verbatim}
psi> typecheck a(x : Base).\overline{hello}<x> | \overline{a}<world> :

Err: The term a is not defined in the type environment!
\end{verbatim}

To circumvent the exception raised above, we need to add the name \( a \) with the appropriate type to the type environment of the running interpreter. To do so we issue the following command:

\begin{verbatim}
psi> deftype { a : "Ch(Base)" ];
\end{verbatim}

Using the above command, we have extended the current type environment with the name \( a \) and its corresponding type \footnote{Note that this types are specific to each instance (pi-calculus instance in this example) and might change from one instance to the other. For more details on types in pi-calculus instance refer to Section 4.2.1} of \( \text{Ch(Base)} \). To see the current type environment, one can issue the following command in the loaded command interpreter.

\begin{verbatim}
psi> tyenv;
\end{verbatim}

\begin{verbatim}
a : "Ch(Base)"
\end{verbatim}

Now that we have the name \( a \) and its \textit{appropriate} type in the type environment, lets try one more time the above agent in the \texttt{typecheck} command and see what happens this time:

\footnote{Note that this types are specific to each instance (pi-calculus instance in this example) and might change from one instance to the other. For more details on types in pi-calculus instance refer to Section 4.2.1}
psi> typecheck a(x : Base),'hello<_x> | 'a<_world> ;

Err: The term hello is not defined in the type environment!

Again type-checker complains; but this time the term hello is the culprit. Note that in the above agent, the term hello is used as a channel to send the name x on it. From the type-checking point of view, the term hello must have the type Ch(Base) in order to be able to serve as a channel to send the name x with the type Base over. We don’t have the possibility to add the explicit type annotations here in the output agent (due to the limitations imposed by the design of the type system of psi-calculi), therefore we must add the term hello and its corresponding type to the type environment. The same statement holds for the term world (it must be inserted into the type environment with its corresponding type which is the Base type in this example).

To extend the type environment with the other two (name,type) pairs we issue the following command in the command interpreter:

psi> deftype { hello : "Ch(Base)" ; world : Base};

psi>

The modified type environment is then as follows:

psi> tyenv;

a : "Ch(Base)"
hello : "Ch(Base)"
world : Base

psi>

Now we are ready to run the typecheck command passing it our parallel agent again:

psi> typecheck a(x : Base),'hello<_x> | 'a<_world> ;

psi>

This time the control silently returns to the command prompt, without any error message. This means that the type-checker succeeded and the agent is well-typed.

Now let us issue a sstep command on the above agent:

45
We are presented with three possible derivatives (this means that the agent has passed the checks by the type-checker and consequently it is well-typed). There is also a new prompt presented to the user which shows the subcommands available inside the sstep command. For full account of the sstep command and its usage, one can refer to [13, Section 3.1.4]. Both the commands sstep and typecheck pass the agent to the type-checker module. The difference between them is that the latter just type-checks the agent and succeeds or raises errors, but the former passes the agent to the evaluator module in case of the success of the type checking; otherwise it raises typing errors and quits.
Here we give a simple example of an agent with bound output and show its type-checking procedure and its valid transitions. Consider the following agent:

$$(\text{new}(b : \text{Base}))(\pi b)$$

Note that name $b$ in the object of the output is bound. If we pass this agent to the `typecheck` command we will have:

```
psi> typecheck (new(b : Base))('a<b>);
```

Err: The term $a$ is not defined in the type environment!

;  

The error is caused by the fact that the name $a$ is not defined in the type environment. So we need to insert the name $a$ with its appropriate type into the environment:

```
psi> deftype { a : "Ch(Base)"};
```

```
psi>
```

Now if we again pass the agent to `typecheck` command

```
psi> typecheck (new(b : Base))('a<b>);
```

```
psi>
```

Type checker issues no error meaning that the agent is well-typed. To see the valid transitions for this agent, now we pass it to `sstep` command:

```
psi> sstep (new (b : Base))('a<b>);
```

1 possible derivative(s)

1 ---

1 |> --|ga(b : Base)b|-->

Constraint:

$$\text{(new b : Base)}/ "a = ga" /$$

Solution:

$$[\text{ga := a}, 1]$$

Derivative:

$$0$$

```
```

And the agent has one possible transition as expected.

Now let us give a sample run which shows the use of the `Invocation` agents. We define the first agent as follows:
In the above command, note that we have given the types of the formal parameters explicitly. This is a requirement on the typed Invocation agents definition. We add another agent definition to the clause environment:

```plaintext
psi> B(ch : "Ch(Base)", world : Base) <= 'ch<world>;
```

After issuing the above commands the current clause environment is as follows:

```plaintext
psi> env;
```

Now we call the defined agents with actual parameters and pass them to the type checker (suppose that we have restarted the tool and type environment is empty):

```plaintext
psi> typecheck A<a,hello> | B<a,world>;
```

Err: The term a is not defined in the type environment!

We also need to extend the type environment with the terms (or names) `hello` and `world` and their appropriate types:
We have all the requirements and we can issue the `typecheck` command as follows (assuming that we already have the definition of the identifiers in the clause environment):

```plaintext
psi> typecheck A<a,hello> | B<a,world>;
```

This time the type-checker does not complain at all (meaning that the agent is well-typed).

Now we turn to show the type-checking of the recursive invocation calls. We try the following simple recursive agent in the `typecheck` command.

```plaintext
D(ch:"Ch(Base)") <= ch(x:Base).D<ch>;
```

The above agent is recursive considering the fact that it invokes itself in its definition. Calling the `typecheck` command with the above agent and the actual parameter `a` yields the following output (assuming that `a` is present in current type environment with an appropriate type):

```plaintext
psi> typecheck D<a>;
```

If we pass the agent `D` to the `sstep` command:

```plaintext
sstep D<a>;
Type <num> for selecting derivative, b - for backtracking, q - quit
1 possible derivative(s)
1 ---
  1 />
    --|ga(x)|--> 
Constraint: 
  \[a = ga\] / 
Solution: 
  ([ga := a], 1)
Derivative: 
  D<a>
```

We have one possible transition and selecting that transition, again we have the same transition and so on. This means that the agent is calling itself recursively:
Note that in all the examples above, we assumed that the types of the names appearing in the agents are compatible wherever it is required. To make this statement clear, let us give an example here. Assume the following simple input agent:

\[ a \in \text{Ch(Base)} \]

Now let us insert the name \( a \) with the type \( \text{Ch(Base)} \) into the type environment:

\[
\text{psi}\text{> deftype \{a:"Ch(Base)"\};}
\]

\[
\text{psi}\text{> typecheck a(x:"Ch(Base)");}
\]

Err: Input type compatibility error!

The type-checker outputs an error message complaining about the use of incompatible types in the input agent. Based on the syntax of the agent, the channel \( a \) is only capable of receiving message of type \( \text{Base} \). But here we are forcing it to receive a name with the type \( \text{Ch(Base)} \) and this incompatibility between the types of the object and subject in the input agent is caught by the type-checker (using instance specific functions 4.2.1). Now we are done with our typed pi-calculus agent simulation.

In next chapter, we discuss the implementation of the typed distributed pi-instance which has structured types and the terms have composite form and are more complex compared to the terms in simply typed pi-calculus instance.
4.2.3 Distributed Pi-calculus instance

In this section we explain and investigate the modeling of the typed distributed pi-calculus instance and implementing it using our typed psi-calculi framework. Before we go through the implementation details, we give a brief introduction to the typed distributed pi-calculus. This calculus was first proposed by Hennessy [14] and the first type system for it is given by Hennessy and Riely [32]. In Dpi (from now on, we call the distributed pi-calculus as Dpi), there is a concept of location. Locations are named. Processes are located in named locations and can move to other named locations. We depict a process at the location \( l \) by \( l[P] \) (network). The networks or locations, can be seen as a named collection of processes; the notion \( l[P] \) shows the process \( P \) located at the network \( l \). The movement of processes from one location to another (migration) is shown by \((\text{go } k.P)\). According to Hüttel, the syntax of processes of typed Dpi is as follows:

\[
P ::= 0 \mid \pi(x).P_1 \mid a(x).P_1 \mid P_1 \mid P_2 \mid (\nu n : T)P_1 \mid !P_1 \mid \text{go } k.P_1
\]

The syntax for the networks is as follows:

\[
N ::= 0 \mid N_1 \mid N_2 \mid (\nu n : T)N_1 \mid l[P]
\]

Networks are ranged over by \( N \) in the above syntax.

There is a translation from Dpi to pi-calculus by Carbone and Maffeis [9], presenting the Dpi as pi-calculus. According to this translation, the set of terms in the calculus is \( T = \{l.a \mid l, a \in \mathcal{N}\} \). This means that we have composite terms in the psi-calculus representation of the Dpi. In the composite term \( l.a \), \( l \) is a location and \( a \) is a channel at that location. we also have the following set of conditions:

\[
C = \{l.a \leftrightarrow l.a \mid l, a \in \mathcal{N}\}
\]

Due to Carbone and Maffeis [9], the translation of the processes from Dpi to the psi-calculus framework is as follows:

\[
\begin{align*}
[\text{go } k.P]_l &= [P]_k \\
[\pi(x).P]_l &= (l.a)(x).[P]_l \\
[P_1 \mid P_2]_l &= [P_1]_l \mid [P_2]_l \\
[!P_1]_l &= ![P_1]_l \\
[(\nu n : T)P]_l &= (\nu n : T)[P]_l
\end{align*}
\]

As in the pi-calculus instance, the only assertion we have in Dpi instance is the trivial assertion \( 1 \); the type environment \( E \) always types \( 1 \), i.e. \( E \vdash 1 \).

According to Hüttel, the type system for Dpi is a type system that assigns types to both locations and channel names. The types in this type system are of the following form:

\[
T ::= \text{Ch}(T) \mid \text{Loc}\{a_i : \text{Ch}(T_i)\}_{i \in I} \mid B
\]
In the above syntax, $I$ is a finite index set and $B$ ranges over a set of base types. A location type, describes the available interface of a location; this means that only specified channel names can be used for communication at a location of this particular type. This statement has been translated into the following type rule for composite terms (due to Hütte):

$$E \vdash l : \text{Loc}\{a_i : \text{Ch}(T_i) \}_{i \in I} \quad E \vdash a_i : \text{Ch}(T_i) \text{ for some } i \in I \quad E \vdash l.a_i : \text{Ch}(T_i)$$

In the rule above, $I$ is a finite set. The compatibility relation is given by $\text{Ch}(T) \leftrightarrow T$.

Now we turn to explain the implementation process of the Dpi instance using the Psi-calculi Workbench. We follow the same style as we did in case of the pi-calculus instance, meaning that we explain the process of implementing the typed Dpi-calculus instance using a step by step fashion.

We start by the DPiInstanceRequirements structure and the changes and modifications required by it in order to reflect the typed Dpi instance implementation. The first change we need to apply is the implementation of the composite terms of Dpi. The set of terms in Dpi instance is defined as follows:

$$T \overset{\text{def}}{=} \mathcal{N} \cup \{l.a \mid l, a \in \mathcal{N}\}$$

This implies that in addition to the set of names as terms (similar to the pi-calculus instance), we have also terms that are composed of names (composite terms). We implement the set of terms as follows:

```plaintext
Listing 17: Implementation of terms in Dpi-calculus instance

```

datatype term = Name of name
              | LocationName of name * name
```

In this implementation the composite terms are named by LocationName.

Next, we need to implement the types for the instance. The difference between types of the simply typed pi-calculus and Dpi-calculus is the location type which was not defined in the pi-calculus instance\(^3\).

We need to translate the location type syntax into a suitably equivalent SML implementation. For this purpose we choose the built-in SML list\(^3\).

---

\(^3\)Note that the other three nominal datatype are exactly the same in Dpi and pi instances, therefore we simply skip their implementation here.
type and we build the location data type based on it. The translation of the types is then as follows:

**Listing 18: Implementation of types in Dpi-calculus instance**

```plaintext
datatype ty = Chan of ty
| Location of (name * ty) list
| Base
```

In the above datatype definition, the location type is a recursive datatype (indicated by the use of `ty` in its body) and we use the built-in list constructor to build the location type. The difficulty that raise here is that in the syntax of the location type, the name `a` that appears in the body of type, must carry only the type of the form `Ch(T)`. We simply neglect this fact here, but later in the implementation of the parser to parse the types, we describe how we have imposed this constraint in the construction of the location types.

Now we turn to define the implementation of the required operators (equivariant functions) for the Dpi instance. For two composite terms to be channel equivalent, the corresponding locations must be equal and also corresponding channels must be equal.

The support function for composite terms is shown in the following listing:

**Listing 19: Implementation of the support function for terms in Dpi-calculus instance**

```plaintext
fun supportT (LocationName(n1, n2)) = [n1, n2]
| supportT (Name n) = [n]
```

The support for a composite term is the union of the supports of its corresponding names. Next function is the `supportTy` function which implements the support operation on types as follows:

**Listing 20: Implementation of support for types in Dpi-calculus instance**

```plaintext
fun supportTy (Chan(t)) = supportTy t
| supportTy (Location([])) = []
| supportTy (Location((n, t)::rs)) =
  (n::supportTy t)@supportTy(Location(rs))
| supportTy (Base) = []
```

To implement the swap operation on terms we do as follows:

**Listing 21: Implementation of swap for terms in Dpi-calculus instance**

```plaintext
fun swapT (Chan(t)) = chan(swapT t)
| swapT (Location(rs)) = Location(swapT(rs))
| swapT (Name n) = Name n
| swapT (Const a) = Const a
```

53
And the swap function for types:

```ml
fun swapTy pi (Chan(t)) = Chan(swapTy pi t)
  | swapTy pi (Location(l)) = Location (unwrap pi l)
  | swapTy pi (Base) = Base
and unWrap pi [] = []
  | unWrap pi ((n,t)::rs) = (swap_name pi n, swapTy pi t)::(unwrap pi rs)
```

Next function is the substitution function which has a different implementation in case of terms and types in typed Dpi instance – comparing to the typed pi-calculus instance. The implementation of the substitution function for composite terms is as follows:

```ml
fun substT sigma (LocationName(n1,n2)) =
  let val f1 = List.find (fn (x,_) ⇒ x = n1) sigma
  val f2 = List.find (fn (x,_) ⇒ x = n2) sigma
  in
    case f1 of
    NONE ⇒ (case f2 of
               NONE ⇒ LocationName(n1,n2)
               SOME (_,t) ⇒ t)
    | SOME (_,t) ⇒ (case f2 of
                   NONE ⇒ t
                   SOME (_,t') ⇒ t')
  end
  | substT sigma (Name a) =
    (case List.find (fn (b,_) ⇒ a = b) sigma of
     NONE ⇒ Name a
     | SOME (_,t) ⇒ t)
```

Next listings summarizes the implementation of the substitution function for types in Dpi instance:

```ml
fun substTy sigma (Chan(t)) = Chan(substTy sigma t)
  | substTy sigma (Location(l)) = Location(unWrap' sigma l)
  | substTy sigma (Base) = Base
```

54
and unWrap' sigma [] = []
| unWrap' sigma ((n,t)::rs) =
  (n, substTy sigma t)::(unWrap' sigma rs)

Now that we are done with the required nominal datatypes implementations and their corresponding equivariant operations. In order to complete the requirements of the DPiInstanceRequirements structure, we turn to implement the functions required by the type-checker (instance specific functions).

As we stated before, these functions are required by the type-checker to type check the terms that appear in the processes. Due to the fact that terms have a different form in each instance (they might be simple names or structured terms etc.), therefore we devolve the implementation of these functions to the user who is going to instantiate a particular calculus, thus providing an instance specific strategy.

The implementation of the checkT function in Dpi instance is as follows:

Listing 25: Implementation of checkT function for in Dpi-calculus instance

fun checkT e (LocationName(l,a)) = checkCompositT e (l,a)
  | checkT e (Name n) =
    (case Env.find e n of
      SOME ty ⇒ ty
    | NONE ⇒ raise TypeErr
      ("The term " ^ n ^ " is not defined in the type environment"))
and checkCompositT e (l,a) =
  let val Tl = checkT e (Name l)
      val Ta = checkT e (Name a)
  in
      case Tl of
        Location(nls) ⇒
          if (member a (Lst.fsl nls)) then Ta
          else raise TypeErr ("Err: Channel " ^ a ^ " is not in the interface of location" ^ l ^ "!")
      | _ ⇒ raise TypeErr ("Err: Name " ^ l ^ " has a malformed type; expected location type!")
    end

We have seen before, that in the Dpi instance a specific location has an interface, meaning that only specified channel names can be used for communication at that location, and we have seen that this requirement is controlled using a type rule for the composite terms. In order to implement this type rule in the Dpi instance, we have introduced a new function checkCompositT. This function is then used by checkT function and its purpose is to implement the type rule for the composite terms.
The next instance specific function related to the type-checker requirements is the checkC function. The implementation is the same as typed pi-calculus instance 4.2.1 therefore we won’t repeat the code here.

So far, we have constructed the DPiInstanceRequirements structure with all the required nominal datatypes and equivariant operations on them as well as the instance specific functions. In order to make the instance work properly, the next step is to define the appropriate parsing and pretty printing functions for the nominal datatypes of the typed Dpi instance.

We start by the implementation of the printer functions. The list below depicts the printer functions in Dpi instance.

**Listing 26: Implementation of printer functions in Dpi-calculus instance**

```plaintext
fun printN a = a
fun printT (LocationName (l, a)) = (printN l) ^ @ ^ (printN a)
   | printT (Name n) = n
fun printC (Eq (a, b)) = (printT a) ^ " = " ^ (printT b)
fun printA psi = "1"
fun printTy (Base) = "Base"
    | printTy (Chan t) = "Ch( " ^ printTy t ^ ")"
    | printTy (Location l) = "Loc{ " ^ printTy l
    and printTy' [] = "}"
    | printTy' ((n, t)::rs) = n ^ ":" ^ printTy t ^ " " ^ printTy' rs
```

The syntax of the terms in Dpi instance can be expressed using the following grammar production rule:

\[
\langle \text{term} \rangle ::= \text{"'} \langle \text{locName} \rangle \text{"'} (\langle \text{identifier} \rangle \text{",} \langle \text{identifier} \rangle \text{"'}\text{"')} | \langle \text{identifier} \rangle
\]

For the complete grammar of the \langle \text{identifier} \rangle, interested reader can refer to the Appendix B.

The implementation of the parser for terms in Dpi instance is as follows:

**Listing 27: Implementation of terms parser in Dpi-calculus instance**

```plaintext
fun term () =
    (Lex.stok "locName" ⇒ Lex.stok "(" ⇒
     name ⇒=
     (fn n1 ⇒ Lex.stok "," ⇒
      name ⇒=
      (fn n2 ⇒ Lex.stok ")" ⇒ Parser.return (LocationName(n1, n2))
     )))
/Lex.identifier ⇒= Parser.return o Name
```
We have used the `choice` combinator in this implementation to make the parser able to parse composite terms as well as simple terms (names). The parser first tries to parse a composite term (we have given those terms a higher priority here) and then (if unsuccessful), it tries to parse a simple term (name). As we have mentioned before, the `choice` combinator has a deterministic behavior.

The next function is the parser function for types. Before listing the code note that the syntax for types in the Dpi instance is based on the following grammar production rules:

\[
\langle \text{location-type} \rangle ::= '\text{Loc}\{' \langle \text{id} \rangle ':' \langle \text{channel-type} \rangle (';' \langle \text{id} \rangle : \langle \text{channel-type} \rangle )^* '}'
\]

\[
\langle \text{channel-type} \rangle ::= '\text{Ch}' \langle \text{dpi-types} \rangle '
\]

\[
\langle \text{base-type} \rangle ::= '\text{Base}'
\]

\[
\langle \text{dpi-types} \rangle ::= \langle \text{location-type} \rangle | \langle \text{channel-type} \rangle | \langle \text{base-type} \rangle
\]

In the above grammar, \( \langle \text{id} \rangle \) stands for identifier and the complete definition is listed in the Appendix B.

The implementation of the parser function for types is then as follows:

```plaintext
fun typ () =
  (Lex.stok "Base" >>= Parser.return Base)
<Parser.choice/>
  (Lex.stok "Ch(" >>=
  (fn t ⇒ Lex.stok ")" >>= Parser.return (Chan t)))
<Parser.choice/>
  (Lex.stok "Loc" ⇒ Lex.stok "{" ⇒
  (nameChannelParse ()) <Parser.sepby/>(Lex.stok ":;"))
⇒
  (fn ntls ⇒ Lex.stok ")" ⇒
  Parser.return (Location(ntls))))

(* Check the wellformedness for Location type while parsing *)
and nameChannelParse () =
  name ⇒
    (fn n ⇒ Lex.stok ":" ⇒ Lex.stok "Ch(" ⇒ (Parser.delayed typ) >>=
    (fn t ⇒ Lex.stok ")" ⇒ Parser.return (n,Chan(t))))
```

One important point here is that, we have deployed an extra function named `nameChannelParse` in the implementation of the `typ()` parser. As we have noted when we gave the implementation of types in Dpi instance, we
need to establish a constraint on the parsing of location types. This function is used by the typ() parser and checks that the user gives the type of the names in locations correctly. The parser only succeeds when the type of a name in a location type is of the form \( \text{Ch}(T) \). This way, we make sure that the location types are constructed correctly. Again, in this parser we use the choice combinator to be able to parse all the three forms of types.

Now we have all the requirements required by a running instance, therefore we construct a structure named D\( \text{pi} \) using the D\( \text{pi} \)Command structure. The D\( \text{pi} \)Command structure is a structure produced by the CommandParser functor, which in turn is the functor that produces the running instance of D\( \text{pi} \). In the next section, we present some sample run session for the typed D\( \text{pi} \) instance.

### 4.2.4 Sample run sessions for D\( \text{pi} \)-calculus instance

In this section, we give an example run session for the distributed pi-calculus instance. The agent we are going to work on is the following:

\[
\text{l.a}(x: \text{Base}) \mid \overline{\text{l.a}}
\]

The above agent is the translation of the following agent in pi-calculus instance into the distributed pi-calculus alternative.

\[
\text{a}(x: \text{Base}) \mid \overline{\text{a}}
\]

In the former agent, \( \text{l.a} \) means that we have a channel named \( \text{a} \) at a location named \( \text{l} \). The reader can refer to the 4.2.3 for a complete description of the translation from D\( \text{pi} \) into the psi-calculus.

Now lets try the above agent in a loaded command interpreter session.

```
psi> typecheck "locName(l,a)"(x:Base) \mid "locName(l,a)"<x>;
```

Err: The term \( \text{l} \) is not defined in the type environment!

```
psi>
```

As you might have guessed, the type-checker raises an error; the reason is that, we don’t have the free name \( \text{l} \) defined in the type environment. Let us add this name with its appropriate type into the type environment with the following command:

```
psi> deftype {l:"Loc\{a:Ch(Base)\}"};
```

Warning, free name in type was not defined in the type environment, added automatically!

```
psi>
```
Note that the type-checker issues a warning, stating that the free name \( a \) at the location \( l \) is not defined in the current type environment and it is added to the type environment automatically. This is a requirement imposed by the well-formedness criteria of the type environment[^4].

The type environment is now as follows:

```
 psi> tyenv:

 a : "Ch(Base)"

 l : "Loc{ a:Ch(Base) }"

 psi>
```

We have the necessary type definitions in the current type environment. Let us try the `typecheck` command again and see what we will get this time.

```
 psi> typecheck "locName(l,a)"(x:Base) | "locName(l,a)<x>;
 psi>
```

No error message and the control is at the `psi` prompt. That means, the agent is passed the type-checker and is well typed.

If we try the same agent in the `sstep` command:

```
 psi> sstep "locName(l,a)"(x:Base) | "locName(l,a)<x>;
 Type <num> for selecting derivative, b - for backtracking, q - quit
 3 possible derivative(s)
 1 ---
    1 />
    --\tau\-->
    Constraint:
    / "l.a = l.a" /
    Solution:
    ([], 1)
    Derivative:
    (0) / (0)

 2 ---
    1 />
    --\ga(\ga)\-->
```

[^4]: According to Hüttel, to keep the type environment well-formed after adding a free name to it, all the free names in the type of that name must already be defined in the current type environment.
This time, we are presented with three possible transitions, meaning that the agent is well-typed and is passed to the evaluator. Note the constraint of the first transition (which is a silent transition). The equality of two composite terms of the two sub agents of the parallel agent must hold to make this transition valid. In the above agent this constraint is satisfied and the transition is eligible. By entering the number 1 in the sstep command, the agents does a \( \tau \) transition and then there is no more transitions available in this case. The other two transitions correspond to the agents acting independently.

This time, let us try a bit more elaborate agent with respect to the location types. We restart the tool and issue the following command in the command interpreter:

```plaintext
psi> deftype {l:Loc{a:Ch(Base); b:Ch(Base)}};
```

Warning, free name in type was not defined in the type environment, added automatically!

```plaintext
psi>
```

The type environment is then:

```plaintext
psi> tyenv;
```

```plaintext
a : "Ch(Base)"

b : "Ch(Base)"
```
l : "Loc a:Ch(Base) b:Ch(Base) "

Note that this time, we have two free names (a and b) appearing inside the definition of the location type of the name l. Both have been added to the type environment with their corresponding types.

Now we try the following agent

\[ l.a(x : \text{Base}) | \overline{b}.x \]

And we pass it to the typecheck command.

\[ \text{psi} > \text{typecheck} \ "\text{locName}(l,a)"(x:\text{Base}) | "\text{locName}(l,b)"<x>; \]

The agent is well-typed! If we pass it to the sstep command:

\[ \text{psi} > \text{sstep} \ "\text{locName}(l,a)"(x:\text{Base}) | "\text{locName}(l,b)"<x>; \]

Type <num> for selecting derivative, b - for backtracking, q - quit
2 possible derivative(s)

1 ---

1 |->

| ga(ga) |

Constraint:
| "l.a = ga" |

Solution:

(1, 1)

Derivative:

("l.b"<x>)

2 ---

1 |->

| ga x |

Constraint:
| "l.b = ga" |

Solution:

(1, 1)

Derivative:

("l.a"(x : Base))

sstep>
This time we are offered two possible transitions. The agents can only transits independently and are not able to interact with each other. This is due to the fact that the constraint which is required by the $\tau$ transition is not satisfied this time ($a \neq b$).

This is the end of the explanation of the typed Dpi instance implementation. We saw how to translate the structured terms when implementing them in the instance. The types of the Dpi instance are more complicate compared to the types of the simply typed pi-calculus. Using this implementation scenario, we have shown how to manipulate types that may contain free names in their structure (in Dpi instance, names appear in types).
5 Conclusion

In this thesis work, we presented the procedure of integrating the type systems framework of Hüttel [17] into the Psi-calculi Workbench.

What we did in this project was to design and implement the type system based on its specification. We modified different parts of the tool in order to provide the necessary prerequisites for the integration of the type system in the tool. We developed a type-checker module that is the translation of the type judgments of the psi-calculi type system into SML code. We also extended the type rules of the psi-calculi type system with a typing rule for the Invocation process and we extended the proof of the subject reduction property to this rule. Extending the command interpreter with the concrete syntax for types – allowing user to enter typed processes – was also a part of the implementation. To investigate the results of our implementation of type system for psi-calculi, we instantiated two different typed calculi. One is the simply typed pi-calculus by Sangiorgi and Walker [33]. And the second one was the typed distributed pi-calculus of Hennessy and Riely [32]. By implementing these two instances in the psi-calculi workbench, we showed how the general framework for type system of psi-calculi can be helpful to study the different typed extensions to the pi-calculus.

5.1 Future work

Developing a complex type system equipped with the features of a fully fledged type system is the potential future works of this project. For example, instantiating the framework with more complex examples and with instances that contain non-trivial assertions and investigating the implementation issues related to such instances is part of the future work of this project. Considering the possibilities of extending the type system of the Psi-calculi Workbench with more general account of sub-typing is also a potential addendum to the project. One other possible future work of this project is to design and implement a type inference system. This way, the users don’t need to are care about the explicit type annotations when issuing a command including an agent at the command interpreter. These are among potential extensions and are parts of possible future work of this project.

5.2 Related work

Other similar works has been targeted the issue of implementing a general type system for process calculi. They are mostly focused on defining a general type system for the pi-calculus; most of them have defined a notion of behavior for the processes. Among them is the work by Honda [16] who introduced typed algebras that can be provided with a notion of reduction.

63
Honda tries to establish a theory of types for concurrent processes which is based on typed algebras. In his work he follows the same principles as those in the study of type disciplines for programming languages – that is based on composing the typed elements in a way which results in well-typed complex programs – and applies it in the context of concurrent computing. According to Honda, a typed algebra is built by starting from a simplest possible algebraic framework which comprises the basic elements of process composition \[16\]. In his paper, Honda shows that his abstract framework is applicable to many different concrete process algebras. Examples such as sorting for polyadic pi-calculus and typed Lafont’s interaction nets \[21\] are given by Honda to show that his results were successful to generalize a type system for a range of different process algebras. His work is different from ours mainly because he focuses on the process composition and type algebras in order to generalize a type theory for concurrent processes where as in our framework the type system does not have such a notion of type and process composition.

Igarashi and Kobayashi \[18\], described a general type system for subset of the polyadic pi-calculus. They showed that this type system can be instantiated to describe the absence of e.g. deadlock and race conditions in the process’s interactions. In their work, the types and type environments has been presented as abstract processes. The main difference between this framework and the Typed Psi-calculi framework is that, in type system for psi-calculi, only terms have type and processes are either well-typed or not (under some well-formed type environment). Therefore the approach taken by the simple type system for psi-calculi is more concise regarding the type system size and complexity.

Bigraphs \[24\], is also aimed to provide a general framework for process calculi. A type system for bigraphs has been proposed by Elsborg et al \[11\]. But in our knowledge, there are not many actual results to show how this type system can be instantiated.

In the work by Makholm and Wells \[22\], they have described a general type system named Poly* for a general process calculus named Meta*. This type system satisfies a subject reduction property. According to Hüttel, the differences from the psi-calculi type system are: the focus of \[22\] is the variants of Mobile Ambient \[10\], not the pi-calculus extensions. The Poly* type system types the processes, not the terms (which are typed in psi-calculi type system) and finally, there is no instantiated example of the Poly* type system; so according to Hüttel, we don’t know how this type system can be instantiated \[17\].

5.3 Acknowledgments

I am grateful to my supervisor Johannes Borgström for his valuable help and guidance during this thesis project. Without his support (both in im-
plementation part as well as the theoretical background), it would be much harder for me to finish this work. He also helped me in the report writing by providing me precious feedback on drafts of this text.

I would like also to give thanks to my reviewer Björn Victor. He kindly provided me valuable guidelines, specially regarding the preparation of the report for the project.

Finally, I appreciate help by Ramunas Gutkovas, who was such a supportive friend during the time I was trying to understand the practical parts of the Psi-calculi Workbench.
References


A Type-checker implementation

Listing 29: Implementation of typing rules (type-checker)

(* The Type-checker is implemented by a functor. *)
(* This functor takes a structure of type FRAME and *)
(* one of type PSI_CLAUSE_ENVIRONMENT and returns a *)
(* structure of type TYPE_CHECKER. *)

signature TYPE_CHECKER =
  sig
    structure Frm = FRAME
    structure ClEnv = PSI_CLAUSE_ENVIRONMENT
  exception Type of string
  val typeCheck : Frm.Psi.Inst.Env.env → ClEnv.env → Frm.Psi
                 .psi → unit
end;

functor TypeChecker (A : sig
structure Fr = FRAME
structure ClEnv = PSI_CLAUSE_ENVIRONMENT
sharing ClEnv.Cl.Psi = Fr.Psi
sharing ClEnv.Cl.Psi.Inst = Fr.Psi.Inst end) : TYPE_CHECKER =
struct
  local open A
  in
    structure Frm = Fr
    structure ClEnv = ClEnv
  end
  exception Type of string
  (* exception FreshNess of string *)

  open Frm
  open ClEnv
  open Missing
  (* aliases *)
  val checkTerm = Psi.Inst.checkT
  val checkCompatIn = Psi.Inst.compatIn
  val checkCompatOut = Psi.Inst.compatOut
  val checkC = Psi.Inst.checkC
  val checkAsser = Psi.Inst.checkA
  val checkPattern = Psi.Inst.checkAbs

  69
val supportTy = Psi.Inst.Type.support
val swapTy = Psi.Inst.Type.swap
val swapAtom = Psi.Inst.Term.swapAtom
val assrSupp = Psi.Inst.Assr.support

val dom = Psi.Inst.Env.getMap
val extendList = Psi.Inst.Env.extendList
val extend = Psi.Inst.Env.extend
val extendA = Psi.Inst.Env.extendAssr

val tyEq = Psi.Inst.Type.eqData

exception TypeErr = Psi.Inst.TypeErr

structure PAlpha = NominalAlphaTyped(Psi)
structure FAlpha = NominalAlpha(Frm)
structure CAlpha = NominalAlpha(Cl)
structure L = NominalList(Psi.Inst.Type)

fun swap [ ] l = l
    | swap (ab::pi) l = 
        let val l' = swap_tvec ab l
        in
            swap pi l'
        end
    and swap_tvec _ [ ] = [ ]
    | swap_tvec ab ((a,t)::rest) = 
        (swapAtom ab a, swapTy ab t)
        ::(swap_tvec ab rest)

(* The typeCheck function implements the typing rules *)
fun typeCheck env clEnv exp =
    case exp of
        Psi.Output (m,n,p) ⇒ checkOutput (m,n,p,env, clEnv)
    | Psi.Input (m,xvec,n,p) ⇒ checkInput (m,xvec,n,p, env,clEnv)
    | Psi.Case clist ⇒ checkCase clist env clEnv
    | Psi.Restriction (n,t,p) ⇒ checkRestriction (n,t,p, env,clEnv)
    | Psi.Parallel (p,p') ⇒ checkParallel (p,p',env, clEnv)
    | Psi.Replication p ⇒ checkReplication (p,env, clEnv)
    | Psi.Assertion a ⇒ checkAssertion a env clEnv
    | Psi.Invocation (a,ml) ⇒ checkInvocation (a,ml,env ,clEnv)
    | Psi.Nil ⇒ ()
and checkOutput (m, n, p, env, clEnv) =
let
  val Ts = checkTerm env m
  handle TypeErr s ⇒ raise Type s
  val To = checkTerm env n
  handle TypeErr s ⇒ raise Type s
  in
  if (checkCompatOut env Ts To) then typeCheck env clEnv p
  else raise Type ("Err: Output type compatibility error!")
  end
and checkInput (m, xvec, n, p, env, clEnv) =
let
  val supAndBndP = Lst.intersection (Psi.support p) (Lst.fsl xvec)
  val supT = List.concat (map (fn t ⇒ supportTy t) (Lst.scl xvec))
  val domE = dom env
  val _ = if ((Lst.intersection domE supT) ≠ supT) then
    raise Type ("Err: Free name of T not in E!")
  else ()
  val sXvec = Lst.intersection domE (Lst.fsl xvec)
  val pi = PAlpha.freshNames domE supAndBndP
  val pi' = PAlpha.freshNames domE sXvec
  val p = PAlpha.permute pi p
  val xvec = swap pi xvec
  val xvec' = swap pi' xvec
  val env' = extendList env xvec'
  val Us = checkTerm env m
  handle TypeErr s ⇒ raise Type s
  val Uo = checkPattern env' xvec' n
  handle TypeErr s ⇒ raise Type s
  in
  if (checkCompatIn env Us Uo) then typeCheck env' clEnv p
  else raise Type ("Err: Input type compatibility error!")
  end
and checkRestriction (n, t, p, env, clEnv) =
let
  (∗ val supAndBndP = Lst.intersection (Psi.support p) (Lst.fsl xvec)
  val supT = supportTy t
  val domE = dom env
  val _ = if ((Lst.intersection domE supT) ≠ supT) then
    raise Type ("Err: Free name of T not in E!")
  else ()
  (* val pi = PAlpha.freshNames domE supAndBndP *)
  (* val p = PAlpha.permute pi p *)
  (* val xvec = swap pi [(n,t)] *)
  val env' = extendList env [(n,t)]
  in
  typeCheck env' clEnv p
  end
and checkReplication \((p,\text{env},\text{clEnv})\) = typeCheck \text{env} \text{clEnv} p

and checkCase \([],\text{env}\) \text{clEnv} = ()
| checkCase \(((c,p)::\text{rest})\) \text{env} \text{clEnv} = 
  let
    val Tc = checkC \text{env} c
    handle TypeErr s ⇒ raise Type s
  val Tp = typeCheck \text{env} \text{clEnv} p
  in
    checkCase \text{rest} \text{env} \text{clEnv}
  end

and checkAssertion a \text{env} \text{clEnv} =
  checkAsser env a handle TypeErr s ⇒ raise Type s

(* To implement this rule, we need the frame of each agent
* freshness conditions:
* 1−dom(E1) # dom(E)
* 2−dom(E2) # dom(E)
* We collect the assertions and bindings occured in the other
* agents
* while type−check the opposite process and vice versa. *)
and checkParallel \((p,q,\text{env},\text{clEnv})\) =
  let
    val domE = dom \text{env}
    (* dom(\text{Ep1}) # dom(\text{E}) *)
    val fQ = Frm.f q
    val bindersFQ = Frm.binders fQ
    val fQNames = (Lst.fsl bindersFQ)
    val ffQ = FA.\text{makeFresh} fQNames fQ domE
    val env' = extendList \text{env} (Frm.binders ffQ)
    val env'' = extendA env' (Frm.\text{assertion} ffQ)
    val _ = typeCheck env'' \text{clEnv} p

    (* dom(\text{Ep1}) # dom(\text{E}) *)
    val fP = Frm.f p
    val bindersFP = Frm.binders fP
    val fPNames = (Lst.fsl bindersFP)
    val ffP = FA.\text{makeFresh} fPNames fP domE
    val env' = extendList \text{env} (Frm.binders ffP)
    val env'' = extendA env' (Frm.\text{assertion} ffP)
    val _ = typeCheck env'' \text{clEnv} q
  in
    ()
  end

(*
* n(\text{P}) \text{subsetequal} xvec
* \mid xvec = \mid Mvec \mid \text{guarded}(\text{P})
* \text{(A, xvec, Tvec, P)} in e
* \text{Psi, e} \mid > P[xvec := Mvec] \rightarrow P'
* Invocation
* \text{Psi, e} \mid > A\text{<Mvec}> \rightarrow P'
*)
and checkInvocation (a, ml, env, clEnv) =
  let
  val clauses = ClEnv.find clEnv a
  val valid = List.filter
    (fn (cl as (a, xtvec, p)) ⇒
      (List.length xtvec = List.length ml)) clauses
  in
  case valid of
  [] ⇒ print ("Err: The clause is not defined in the
              clause environment! \n")
  | _ ⇒ let
    val mtvec = map (fn m ⇒ checkTerm env m handle
                   TypeErr s ⇒ raise Type s) ml
    val tyvec = List.concat ($ map supportTy mtvec
    val _ = map (fn (cl as (a, xtvec, _)) ⇒
                 CAlpha.makeFresh (List.fsl xtvec) cl tyvec) valid
    val _ = map (fn (a, xtvec, _) ⇒
                 let val tvec = List.scl xtvec
                       in
                       if not (List.all tyEq (List.zip tvec mtvec))
                       then
                         raise Type ("Err: Type error in
                                     identifier arguments!")
                       else ()
                       end)
    in
    ()
  end
end

Listing 30: The TYPE_ENVIRONMENT signature

signature TYPE_ENVIRONMENT =
sig
  type env
  type name
  type typ
type assr

val find : env → name → typ option
val extend : env → name → typ → env
val extendl : env → (name * typ) list → env
val extendAssr : env → assr → env
val getMap : env → name list
val getTEnv : env → (name * typ) list
val getAssr : env → assr
val empty : env

end

Listing 31: ORD_KEY signature used by TYPE_ENVIRONMENT signature

signature ORD_KEY =
sig
  structure Assr: NOMINAL
  structure Type: NOMINAL

  sharing type Assr.atom = Type.atom
val compose : Assr.data → Assr.data → Assr.data
val unitAssr : Assr.data

end

Listing 32: TypeEnv functor

functor TypeEnv (Key : ORD_KEY) : TYPE_ENVIRONMENT =
struct
  local
    open Lst
    open Tpl
  in
    type name = Key.Assr.atom
    type typ = Key.Type.data
    type asser = Key.Assr.data
    type env = ((name * typ) list * asser)

    fun find (e,_) k = Lst.assoc k e
    fun ext (k,v) e =
      case Lst.assoc k e of
        SOME _ ⇒ (print ("\nWarning: Name already in the domain of E!\n\n")); e)
      | NONE ⇒ let val nameInTyInE = intersection (Key.Type.
                     support v) (fsl e)
in
  if (nameInTyInE \neq (Key.Type.support v))
    then
      (print ("\nErr: Names in type not already in
          the domain of E!\n\n") ; e)
    else
       (k,v)::e

end

fun extend (e,a) k v = (ext (k,v) e, a)

fun extendl (e,a) kvs = (foldl ext e kvs, a)

fun extendAssr (e,a) assr =
  let val nameInAssrInE = intersection (Key.Assr.support assr) (fsl e)
  in
    if (nameInAssrInE \neq (Key.Assr.support assr)) then
      (print ("\nErr: Names in assertion not already in the
          domain of E!\n\n") ; (e,a))
    else
       (e, Key.compose a assr)
  end

fun getMap (e,_) = fsl e

fun getTEnv (e,_) = e

fun getAssr (_,a) = a

val empty = ([], Key.unitAssr)
end
end
B  Psi Workbench grammar (with types)

B.1 Command grammar

\[
\langle\text{script}\rangle ::= \langle\text{clause}\rangle^*
\]
\[
\langle\text{clause}\rangle ::= \langle\text{term}\rangle \langle\text{clause-args}\rangle \langle\text{tr}\rangle
\]
\[
\langle\text{clause-args}\rangle ::= \langle\text{term}\rangle \langle\text{name-type-list}\rangle \langle\text{tr}\rangle
\]
\[
\langle\text{term}\rangle ::= \langle\text{name}\rangle \langle\text{type}\rangle|\varepsilon
\]
\[
\langle\text{name-type-list}\rangle ::= \varepsilon | \langle\text{name}\rangle \langle\text{type}\rangle\langle\text{name}\rangle \langle\text{type}\rangle\ast
\]
\[
\langle\text{tr}\rangle ::= \langle\text{case}\rangle | \langle\text{assertion}\rangle | \langle\text{nil}\rangle | \varepsilon
\]
\[
\langle\text{case}\rangle ::= \langle\text{cond}\rangle \langle\text{agent}\rangle
\]
\[
\langle\text{assertion}\rangle ::= \langle\text{assr}\rangle
\]
\[
\langle\text{nil}\rangle ::= \langle0\rangle
\]
\[
\langle\text{invocation}\rangle ::= \langle\text{identifier}\rangle \langle\text{term-sequence}\rangle
\]
\[
\langle\text{term-sequence}\rangle ::= \varepsilon | \langle\text{term}\rangle \langle\text{term}\rangle\ast
\]
\[
\langle\text{name}\rangle ::= \langle\text{literal}\rangle
\]
\[
\langle\text{term}\rangle ::= \langle\text{literal}\rangle
\]

B.2 Agent grammar

\[
\langle\text{agent}\rangle ::= \langle\text{parallel}\rangle | \langle\text{restriction}\rangle | \langle\text{replication}\rangle | \langle\text{parens}\rangle | \langle\text{prefix}\rangle | \langle\text{case}\rangle | \langle\text{assertion}\rangle | \langle\text{nil}\rangle | \varepsilon
\]
\[
\langle\text{parallel}\rangle ::= \langle\text{agent}\rangle | \langle\text{agent}\rangle \langle\text{agent}\rangle
\]
\[
\langle\text{restriction}\rangle ::= \langle\text{new}\rangle \langle\text{name}\rangle \langle\text{name}\rangle \langle\text{type}\rangle \langle\text{name}\rangle \langle\text{name}\rangle \langle\text{type}\rangle\ast
\]
\[
\langle\text{parens}\rangle ::= \langle\text{agent}\rangle
\]
\[
\langle\text{prefix}\rangle ::= \langle\text{input}\rangle | \langle\text{output}\rangle
\]
\[
\langle\text{prefix}\rangle ::= \langle\text{input}\rangle \langle\text{term}\rangle
\]
\[
\langle\text{input}\rangle ::= \langle\text{term}\rangle \langle\text{name}\rangle \langle\text{type}\rangle \langle\text{name}\rangle \langle\text{type}\rangle\ast
\]
\[
\langle\text{output}\rangle ::= \langle\text{term}\rangle \langle\text{name}\rangle \langle\text{type}\rangle \langle\text{name}\rangle \langle\text{type}\rangle\ast
\]
\[
\langle\text{case}\rangle ::= \langle\text{condition}\rangle \langle\text{agent}\rangle
\]
\[
\langle\text{assertion}\rangle ::= \langle\text{assr}\rangle
\]
\[
\langle\text{nil}\rangle ::= \langle0\rangle
\]
\[
\langle\text{invocation}\rangle ::= \langle\text{identifier}\rangle \langle\text{term-sequence}\rangle
\]
\[
\langle\text{term-sequence}\rangle ::= \varepsilon | \langle\text{term}\rangle \langle\text{term}\rangle\ast
\]
\[
\langle\text{name}\rangle ::= \langle\text{literal}\rangle
\]
\[
\langle\text{term}\rangle ::= \langle\text{literal}\rangle
\]

76
\( (\text{cond}) \quad ::= \quad \langle \text{literal} \rangle \)

\( (\text{assr}) \quad ::= \quad \langle \text{literal} \rangle \)

\( (\text{type}) \quad ::= \quad \langle \text{literal} \rangle \)

\( (\text{literal}) \quad ::= \quad 'n' \ldots 'n'
\| \quad ':' \ldots ':'
\| \quad '{' \ast \ldots '}'
\| \quad \langle \text{identifier} \rangle \)

\( (\text{identifier}) \quad ::= \quad \langle \text{id} \rangle^{+} \ast \ast \)

\( (\text{id}) \quad ::= \quad \langle \text{alpha-numeric} \rangle \mid '._' \)

\( (\text{alpha-numeric}) \quad ::= \quad [a-zA-Z0-9] \)