Universidade dos Açores
Departamento de Matemática

A compiler for the π-Calculus:

the frontend

Tiago Cogumbreiro

Orientador
Doutor Francisco Cipriano da Cunha Martins

February 2, 2007

Acknowledgements

I wish to express a big thank you to my tutor Professor Francisco Martins. His rigour and experience made me a better scholar, on more fields than computer science, like writing better English, and for sprouting my interest for investigation. Without him this work could not be possible.

A sincere word of appreciation to Professor Vasco Vasconcelos, for the opportunity and for the conditions he created.

To the Departamento de Informática da Universidade de Lisboa for providing the conditions necessary for the realization of this work. To the groups LabMOL and NLX, of that university, for the friendly reception and integration.

I would like to thank the Departamento de Matemática da Universidade dos Açores, for the flexibility of allowing me to assist the classes from abroad.

Finally, I wish to thank the Centro de Investigação em Informática e Tecnologias da Informação, of the Universidade Nova de Lisboa, for the finnancial support.

Contents

Chapter 1

Abstract

Chip multiprocessors is part of an emerging market. Parallel and concurrent programming needs to be mastered in order to take advantage of chip multiprocessors' architecture [16]. The π-calculus is a process algebra that reasons about concurrent interactive systems [15] and is promoted as a foundation for concurrency theory.

We present the frontend of a compiler for a language based on the π-calculus. This compiler is useful for putting in practice the expressiveness brought by the π-calculus.

Our work is divided into two parts: the first part introduces the π-calculus, while the second part provides a practical approach on the construction of a compiler. To develop the compiler we follow the design presented in [1].

The next chapter describes the π-calculus. We study its motivation, its use, and its rules. In the third chapter we talk about the compiler. We analyse each step taken and the choices of design when building the compiler. Finally, the closing chapter discusses future work, namely the backend of the compiler, targeted on a multithreaded typed assembly language.

Chapter 2

The π-Calculus

2.1 Introduction

A process is a sequence of actions performed by a system, be it software, a machine, or a human being. Processes also stand for the behaviour of a system. A process algebra is a formal mathematical system consisting of a set of processes and operations on those processes [2].

Computational behaviour was initially conceived considering a simple model: to regard a behaviour as an input/output function. This model of behaviour is present on the Turing machine [20], register machines and the lambda calculus. Register machines and the Turing machine represent input and output by reading and writing on memory cells. On the lambda calculus the input is provided as an argument to a function and the output is its return value.

Modern computing became more complex. The complexity growth is associated with the algorithms being treated and with the increasing number of systems interacting. Process algebra models behaviour through interaction, rigorously defining the behaviour of concurrent interactive systems, something impossible with the former model of behaviour.

The π-calculus, developed by Robin Milner, Joachim Parrow, and David Walker [15], is a process algebra that describes *mobility*. The π-calculus is used to model a network of interconnected processes interacting amongst each other through connection links. In the π-calculus, processes communicate by sending and receiving references to other processes, allowing the dynamic reconfiguration of the network.

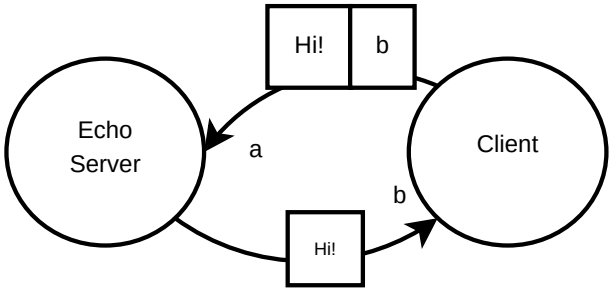

Figure 2.1: Communication with an echo server

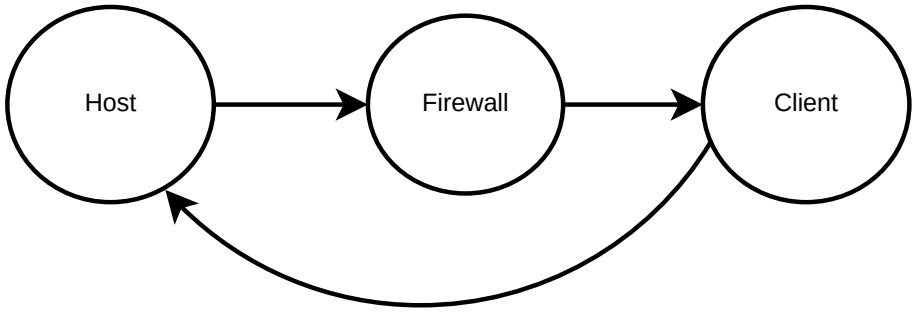

Figure 2.2: Communication with a host behind a firewall

When process A receives a reference to process B, we say that process B is *moving* towards process A. This *movement* characterizes the π-calculus as a *mobile* process algebra.

As an example, consider an echo server, which bounces every message the client sends it. Figure 2.1 illustrates the communication between an echo server and a client. The `echo` server receives a message, through port `a`, containing the text `'Hi!'` and a reference where it should echo the message to the client, the port `b`. Afterwards, the server sends the text `'Hi!'` back to the client, trough port `b`.

Another example is a firewall that can be configured to forward packets from the internet to a computer inside a LAN (Local Area Network), which we refer as the host. The host moves from the LAN to the Internet, when the firewall exposes its link to a computer on the Internet. This scenario can be easily modeled by the π-calculus and it is illustrated in Figure 2.2.

			Values
v	::=	x, a	name
	|	*basval*	basic value
			Processes
P, Q	::=	$\mathbf{0}$	nil
	|	$\overline{x}\langle\vec{v}\rangle$	output
	|	$x(\vec{a}).P$	input
	|	$P \mid Q$	parallel
	|	$(\nu\, x\colon (\vec{T}))\, P$	restriction
	|	$!P$	replication

The syntax of T is illustrated in Figure 2.4

Figure 2.3: Process syntax

2.2 The π-Calculus Syntax

2.2.1 Syntax of Processes

The adopted π-calculus syntax is based on [14] with extensions presented in [18]: typed, asynchronous, and polyadic.

The syntax is divided into two categories: *names* and *processes*. Names are denoted by lower case letters: x, a. Values, v, can symbolise either a name or a primitive value. Processes, denoted by upper case letters, have their syntax depicted in Figure 2.3. The vector above a name or a type abbreviates a possibly empty sequence of names or types (e.g. $\vec{a}$ stands for $a_0, a_1, \ldots, a_n$).

We start by discussing an informal semantics of the π-calculus. The main goal of the calculus is to describe communications between processes. The building blocks of such interactions are input processes, output processes, and names. Processes may be grouped as elementary and composite. This distinction helps understanding how process expressions are built. Elementary processes comprise the nil process, $\mathbf{0}$, corresponding to the inactive process, and the output process, $\overline{x}\langle\vec{v}\rangle$, outlining the action of sending data,

$\vec{v}$, to another process through a channel, x. Composite processes consists of the input, the parallel, the restriction, and the replication. By composing hierarchies and sending links throughout the network we can express how the network reacts to interactions between processes.

The input process, $x(\vec{a}).P$, can receive a sequence of values via channel x and continue as P, with the received names substituted for the received values. The parallel composition process, $P \mid Q$, represents two active processes that run concurrently. The restriction process, $(\nu\, x\colon (\vec{T}))\, P$, is used to create a new channel definition that can only be used inside process P. The replicated process, $!P$, represents an infinite number of active processes composed with the parallel operator.

The following example, is a possible implementation of the echo server depicted in Figure 2.1.

$$!echo(msg, reply).\overline{reply}\langle msg\rangle \tag{2.1}$$

This process is ready to receive a message, msg, and a communication channel, $reply$. After receiving the values, it echoes the message through the reply channel.

The firewall example, illustrated in Figure 2.2, can be expressed as

$$\begin{aligned} &!host(y).P \mid !firewall(c).\overline{c}\langle host\rangle \mid \\ &\overline{firewall}\langle client\rangle \mid client(x).\overline{x}\langle data\rangle \end{aligned} \tag{2.2}$$

The host that resides inside the firewall is analogous to the process $!host(y).P$. We use the replication process to be able to communicate with multiple clients. The client must send a message, $\overline{firewall}\langle client\rangle$, to the firewall, $!firewall(c).\overline{c}\langle host\rangle$, to obtain the reference to the host. In this example, the client simply sends a chunk of data to the host after receiving the channel to the host, $client(x).\overline{x}\langle data\rangle$.

2.2.2 Syntax of types

Types enforce correctness to languages by imposing constraints [5]. Languages that guarantee type consistency at run-time are called strongly typed. If, just by analysing the code, we can extract the type of every expression, the language is statically typed. Statically typed languages are, therefore, strongly typed. With static typing, typing errors are found at compile time,

$$
\begin{array}{rcll}
 & & & \textit{Types} \\
T, S & ::= & B & \text{basic value type} \\
 & | & (\vec{T}) & \text{link type} \\
 & & & \\
 & & & \textit{Basic value types} \\
B & ::= & \mathit{int} & \text{integer type} \\
 & | & \mathit{str} & \text{string type} \\
 & & & \\
 & & & \textit{Type environments} \\
\Gamma & ::= & \Gamma, x\colon T & \\
 & | & \varnothing &
\end{array}
$$

Figure 2.4: Type syntax

leading to safer code being developed. Monomorphic languages ensure that every symbol (i.e., a variable) is of only one type.

Our language is monomorphic and statically typed. We assign types to channels and to basic values. A basic value type is either a string, *str*, or an integer, *int*; the channel type describes types of the value communicated through the channel, composed by a sequence of types. For example, the definition of a channel that may carry an empty tuple:

$$()$$

If we look back at Process 2.1, we can try to infer the type of each argument of the input channel

$$\underbrace{\mathit{echo}(\mathit{msg}, \mathit{reply}).}\, \overline{\mathit{reply}}\langle \mathit{msg} \rangle \tag{2.3}$$

The channel *echo* has two arguments: *msg* and *reply*, thus its type must also have two arguments, let us name them T_1 and T_2. The channel *echo* will be typified by

$$(T_1, T_2)$$

The second value being communicated, the name *reply*, is a channel that outputs a string.

$$\overline{\mathit{reply}}\langle \mathit{msg} \rangle$$

So, T_2 must be of channel type, with one argument of type string

$$(str)$$

and the type of the channel *echo* can be rewritten as

$$(T_1, (str))$$

The first argument of the channel *echo*, the name msg, is sent through the channel *reply*, hence T_1 must be of type string, str. Hence, the name *echo* of the Process 2.3 is of type:

$$(str, (str))$$

2.3 Semantics

The semantics of the π-calculus allows us to express the behaviour of a process. With a rigorous semantics we can identify if two processes have the same structural behaviour, observe how a process evolves as it interacts, and analyse how links move from one process to another.

For clarity's sake, we omit the type from the restriction operator.

2.3.1 Structural Congruence

The syntax differentiates processes that, intuitively, represent the same behaviour [17]. For example, process $P \mid Q$ and process $Q \mid P$ are syntactically different, although our intuition about concurrent processes is that order does not matter. The process $(\nu\, x)\, \overline{y}\langle x \rangle$ and the process $(\nu\, z)\, \overline{y}\langle z \rangle$ are also syntactically different, but both processes represent an output channel sending a private channel that communicates an empty tuple, regardless of the different choice of names.

The *structural congruence* relation, $\equiv$, is the smallest congruence relation on processes closed under rules given in Figure 2.5. Structural congruence allows to identify processes that represent the same behaviour structure. Structural congruence can be used to transform the structure of a process.

The bound name function can be inductively defined as

(S1) change of bound names (see Figure 2.3.1)

(S2) $P \mid \mathbf{0} \equiv P$

(S3) $P \mid Q \equiv Q \mid P$

(S4) $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

(S5) $(\nu\, x\colon (\vec{T}))\,(P \mid Q) \equiv P \mid (\nu\, x\colon (\vec{T}))\,Q \quad \text{if } x \notin \mathrm{fn}(P)$ (see Figure 2.3.2)

(S6) $(\nu\, x\colon (\vec{T}))\,\mathbf{0} \equiv \mathbf{0}$

(S7) $(\nu\, x\colon (\vec{T}))\,(\nu\, y\colon (\vec{S}))\,P \equiv (\nu\, y\colon (\vec{S}))\,(\nu\, x\colon (\vec{T}))\,P$, if $x \neq y$

(S8) $!P \equiv P \mid !P$

Figure 2.5: Structural congruence rules

Definition 2.3.1 (The bound name function)**.**

$$\begin{aligned}
\mathrm{bn}(\mathbf{0}) &= \varnothing \\
\mathrm{bn}(P \mid Q) &= \mathrm{bn}(P) \cup \mathrm{bn}(Q) \\
\mathrm{bn}(x(\vec{a}).P) &= \{\vec{a}\} \cup \mathrm{bn}(P) \\
\mathrm{bn}(\overline{x}\langle\vec{v}\rangle) &= \varnothing \\
\mathrm{bn}((\nu\, x\colon (\vec{T}))\,P) &= \{x\} \cup \mathrm{bn}(P)
\end{aligned}$$

The free name function can also be defined inductively:

Definition 2.3.2 (The free name function)**.**

$$\begin{aligned}
\mathrm{fn}(\mathbf{0}) &= \varnothing \\
\mathrm{fn}(P \mid Q) &= \mathrm{fn}(P) \cup \mathrm{fn}(Q) \\
\mathrm{fn}(x(\vec{a}).P) &= \{x\} \cup \mathrm{fn}(P) \\
\mathrm{fn}(\overline{x}\langle\vec{v}\rangle) &= \{x, \vec{v}\} \\
\mathrm{fn}((\nu\, x\colon (\vec{T}))\,P) &= \mathrm{fn}(P) \backslash \{x\}
\end{aligned}$$

The rule S1 says that a change of bound names maintains structural congruence. The bound names of Process 2.3 are:

$$\mathrm{bn}(\mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle\mathit{msg}\rangle) = \{\mathit{msg}, \mathit{reply}\}$$

Now we rename each bound name: msg by x and $reply$ by y

$$echo(msg, reply).\overline{reply}\langle msg\rangle\{msg/x\}\{reply/y\} \text{ is } echo(x, y).\overline{y}\langle x\rangle$$

If we apply rule S1, the process affected by the change of bound names, $echo(x, y).\overline{y}\langle x\rangle$, is structurally congruent with the original:

$$echo(x, y).\overline{y}\langle x\rangle \equiv echo(msg, reply).\overline{reply}\langle msg\rangle$$

Rule S2 asserts that the nil process is neutral concerning the parallel operator. Rule S3 and rule S4 bring reflexivity and associativity to the parallel process. Rule S5, we can remove a process, P, from within a restriction, $(\nu\, x\colon (\vec{T}))\,(P \mid Q)$, and run it in parallel with that restriction, $P \mid (\nu\, x\colon (\vec{T}))\,Q$, if the name of the new channel, x, is free in P. It also sets forth the opposite: we can move a process that runs in parallel with the restriction, inside of that restriction. The restriction process

$$(\nu\, b)\,(a().\mathbf{0} \mid b().\mathbf{0}) \tag{2.4}$$

is composed by two processes: $a().\mathbf{0}$ and $b().\mathbf{0}$. The process on the right, $a().\mathbf{0}$, has no occurrences of the name b, hence we can rearrange it to the outside of the restriction process:

$$a().\mathbf{0} \mid (\nu\, b)\, b().\mathbf{0} \equiv (\nu\, b)\,(a().\mathbf{0} \mid b().\mathbf{0})$$

These two processes, however, are not structurally congruent:

$$b().\mathbf{0} \mid (\nu\, b)\, a().\mathbf{0} \not\equiv (\nu\, b)\,(a().\mathbf{0} \mid b().\mathbf{0})$$

The name b is free in the process $b().\mathbf{0}$, thus we cannot rearrange it to the inside of a restriction process that is abstracting a channel named b. To perform this transformation we need to apply rule S1 to rename the name b of the input process, $b().\mathbf{0}$, to a name not free in $(\nu\, b)\, a().\mathbf{0}$.

Rule S6 and rule S7 are related to the restriction operator; rule S8 is related to the replicated operator. With rule S6 we discard or add the restriction operator to the nil process. Two consecutive restriction operands, that target different names, are interchangeable, if we use rule S7. Rule S8, unfolds copies of the process being replicated.

The next example illustrates two processes and verifies their structural congruence. The process

$$echo(msg, reply).\overline{reply}\langle msg\rangle \mid (\nu\, r)\,\overline{echo}\langle \text{'}\texttt{hello world!}\text{'}, r\rangle \tag{2.5}$$

and the process

$$(\nu\, y)\,(echo(x,y).\overline{y}\langle x\rangle \mid (\mathbf{0} \mid \overline{echo}\langle \texttt{'hello world!'}, y\rangle)) \tag{2.6}$$

behave the same. Each process represents a client that sends a message to an echo server and, afterwards, discards the reply from the server. We will use structural congruence to transform Process 2.6 into Process 2.5:

$$\begin{aligned}
&(\nu\, y)\,(echo(x,y).\overline{y}\langle x\rangle \mid (\underbrace{\mathbf{0} \mid \overline{echo}\langle \texttt{'hello world!'}, y\rangle}_{\text{Step 1}})) && \text{S3}\\
\equiv\ &(\nu\, y)\,(echo(x,y).\overline{y}\langle x\rangle \mid (\overline{echo}\langle \texttt{'hello world!'}, y\rangle \underbrace{\mid \mathbf{0}}_{\text{Step 2}})) && \text{S4}\\
\equiv\ &(\nu\, y)\,(\underbrace{echo(x,y).\overline{y}\langle x\rangle}_{\text{Step 3}} \mid \overline{echo}\langle \texttt{'hello world!'}, y\rangle) && \text{S5}\\
\equiv\ &\underbrace{echo(x,y).\overline{y}\langle x\rangle}_{\text{Step 4}} \mid (\nu\, y)\,\overline{echo}\langle \texttt{'hello world!'}, y\rangle && \text{S1}\\
\equiv\ &echo(msg,reply).\overline{reply}\langle msg\rangle \mid \underbrace{(\nu\, y)\,\overline{echo}\langle \texttt{'hello world!'}, y\rangle}_{\text{Step 5}} && \text{S1}\\
\equiv\ &echo(msg,reply).\overline{reply}\langle msg\rangle \mid (\nu\, r)\,\overline{echo}\langle \texttt{'hello world!'}, r\rangle
\end{aligned}$$

In the first step we rearrange the process $\mathbf{0} \mid \overline{echo}\langle \texttt{'hello world!'}, y\rangle$, by using rule S3, to remove the needless inactive process, step two. In the process $echo(x,y).\overline{y}\langle x\rangle$, the name y is bound. On step 4 we apply rule S5 to move the input process out of the restriction process, because y is not a free name in the moved process. On the last two steps we apply two changes of bound names and obtain Process 2.5.

2.3.2 Reaction Rules

The π-calculus models computational behaviour through interaction, as usual in all process algebras. Each computational step corresponds to processes' interaction.

The reduction relation $\rightarrow$, defined over processes, in Figure 2.6, establishes how a computational step transforms a process [13]. The formula $P \rightarrow Q$ means that process P can interact and evolve (reduce) to process Q.

The axiom React is the gist of the reaction rules, it represents the communication along a channel [12]. An output process, $\overline{x}\langle\vec{v}\rangle$, can interact with

$$\frac{}{x(\vec{a}).P \mid \overline{x}\langle\vec{v}\rangle \mid Q \to P\{\vec{v}/\vec{a}\} \mid Q}\ \textsc{React}$$

$$\frac{P \to P'}{P \mid Q \to P' \mid Q}\ \textsc{Par} \qquad \frac{P \to P'}{(\nu\, x\colon (\vec{T}))\, P \to (\nu\, x\colon (\vec{T}))\, P'}\ \textsc{Res}$$

$$\frac{P \to P' \quad P \equiv Q \quad P' \equiv Q'}{Q \to Q'}\ \textsc{Struct}$$

Figure 2.6: Reaction Rules

an input process, $x(\vec{a}).P$, if they have the same channel's name, x. The output message, $\vec{v}$, moves along channel x to process P and replaces the entry points, $\vec{a}$, resulting $P\{\vec{v}/\vec{a}\}$. The term $P\{\vec{v}/\vec{a}\}$ means that the names $\vec{a}$, in process P, are to be replaced by the values $\vec{v}$. The process

$$a(x).\overline{x}\langle\rangle \mid \overline{a}\langle y\rangle \mid y().\mathbf{0}$$

is ready to react. We apply rule React and obtain the reduction

$$a(x).\overline{x}\langle\rangle \mid \overline{a}\langle y\rangle \mid y().\mathbf{0} \to \overline{y}\langle\rangle \mid y().\mathbf{0}$$

The application of this rule sets forth the communication between process $a(x).\overline{x}\langle\rangle$ and process $\overline{a}\langle y\rangle$, where name y is sent through channel a.

The remaining three rules are: Par, Res, and Struct. Rule Par expresses that reduction can appear on the right side of a parallel composition. Res rules that reduction can occur inside the restriction operator. Rule Struct brings congruence rules to the reduction relation.

It is important to understand the relevance of structural congruence to the reaction rules. With structural congruence, we are able to reorder and rearrange processes, so that they can react. Structural congruence and process reduction also bring non-determinism to the π-calculus, because we can arrange different processes to react differently.

We illustrate the echo server's communication with a client. The process

$$\mathit{echo}(\mathit{msg}, \mathit{reply}).\overline{\mathit{reply}}\langle\mathit{msg}\rangle \mid (\nu\, r)\, \overline{\mathit{echo}}\langle\texttt{'hello world!'}, r\rangle$$

represents, respectively, the echo server being run concurrently with a client that creates a new name sends it, along with a message, through channel

echo, to the server. The following steps describe the reaction between both processes:

$$
\begin{aligned}
& \underbrace{echo(msg, reply).\overline{reply}\langle msg\rangle}_{\text{Step 1}} \mid \\
& (\nu\, r)\, \overline{echo}\langle \texttt{'hello world!'}, r\rangle && \text{S5} \\
\equiv\ & (\nu\, r)\, (echo(msg, reply).\overline{reply}\langle msg\rangle \mid \\
& \underbrace{\overline{echo}\langle \texttt{'hello world!'}, r\rangle}_{\text{Step 2}}) && \text{S3} \\
\equiv\ & (\nu\, r)\, (echo(msg, reply).\overline{reply}\langle msg\rangle \mid \\
& \overline{echo}\langle \texttt{'hello world!'}, r\rangle \mid \mathbf{0}) && \textsc{Res}, \textsc{React} \\
\rightarrow\ & (\nu\, r)\, (\overline{r}\langle \texttt{'hello world!'}\rangle \mid \mathbf{0}) && \text{S3} \\
\equiv\ & (\nu\, r)\, \overline{r}\langle \texttt{'hello world!'}\rangle
\end{aligned}
$$

We want the restriction to be the outer process, so client and server communicate. Since the name r is free in the process $echo(msg, reply).\overline{reply}\langle msg\rangle$, we can move the restriction to the root of the process tree, using the structural congruence rule S5. Rule Res defines that the reduction can happen inside the restriction process, hence we apply the rule React, then communication takes place. Finally we discard the inactive process, with the application of rule S3.

By using rules S5 and S3, from structural congruence, and rules React and Res, from process reaction, we have illustrated the interaction between the server and the client. The resulting process, $(\nu\, r)\, \overline{r}\langle \texttt{'hello world!'}\rangle$, is a pending output process that contains the message we first sent to the echo server.

The next example is the application of reaction rules to the the Pro-

cess 2.6.

$$
\begin{aligned}
& (\nu\, y)\, (echo(x,y).\overline{y}\langle x\rangle \mid (\underbrace{\mathbf{0} \mid \overline{echo}\langle \texttt{'hello world!'}, y\rangle}_{\text{Step 1}})) && \text{S4} \\
\equiv\ & (\nu\, y)\, (\underbrace{echo(x,y).\overline{y}\langle x\rangle \mid \overline{echo}\langle \texttt{'hello world!'}, y\rangle \mid \mathbf{0}}_{\text{Step 2}}) && \textsc{Res}, \textsc{React} \\
\rightarrow\ & (\nu\, y)\, (\overline{y}\langle \texttt{'hello world!'}\rangle \underbrace{\mid \mathbf{0}}_{\text{Step 3}}) && \text{S3} \\
\equiv\ & \underbrace{(\nu\, y)\, \overline{y}\langle \texttt{'hello world!'}\rangle}_{\text{Step 4}} && \text{S1} \\
\equiv\ & (\nu\, r)\, \overline{r}\langle \texttt{'hello world!'}\rangle
\end{aligned}
$$

On the first step, we rearrange the inactive process, using the structural congruence rule S4, so that the input and the output processes can react. On step two, we apply rule Res and rule React to generate communication between processes, inside the restriction process. On step three, with the application of rule S3, we discard the inactive process. Then, we change the bounded name y into r, with rule S1. The final process, $(\nu\, r)\, \overline{r}\langle \texttt{'hello world!'}\rangle$, is the same process from the previous example.

What happens if we have two processes available to communicate with the echo server? Let us examine this process:

$$
echo(x,y).\overline{y}\langle x\rangle \mid \overline{echo}\langle \texttt{'Proc 1'}, a\rangle \mid \overline{echo}\langle \texttt{'Proc 2'}, b\rangle \tag{2.7}
$$

It is ready for reduction:

$$
\begin{aligned}
& echo(x,y).\overline{y}\langle x\rangle \mid \overline{echo}\langle \texttt{'Proc 1'}, a\rangle \mid \overline{echo}\langle \texttt{'Proc 2'}, b\rangle && \textsc{React} \\
\rightarrow\ & \overline{a}\langle \texttt{'Proc 1'}\rangle \mid \overline{echo}\langle \texttt{'Proc 2'}, b\rangle
\end{aligned}
$$

Now let us use the initial process, Process 2.7, and rearrange the output processes, by using structural congruence rule S4. The resulting process is also ready for reduction:

$$
\begin{aligned}
& echo(x,y).\overline{y}\langle x\rangle \mid \overline{echo}\langle \texttt{'Proc 1'}, a\rangle \mid \overline{echo}\langle \texttt{'Proc 2'}, b\rangle && \text{S4} \\
\equiv\ & echo(x,y).\overline{y}\langle x\rangle \mid \overline{echo}\langle \texttt{'Proc 2'}, b\rangle \mid \overline{echo}\langle \texttt{'Proc 1'}, a\rangle && \textsc{React} \\
\rightarrow\ & \overline{b}\langle \texttt{'Proc 2'}\rangle \mid \overline{echo}\langle \texttt{'Proc 1'}, a\rangle
\end{aligned}
$$

This example clearly shows the underteministic nature of the π-calculus. The same process, after reduction, can generate more then one process.

$$\frac{baseval \in B}{\Gamma \vdash baseval : B}\ \textsc{Tv-Base} \qquad \frac{}{\Gamma, x : T \vdash x : T}\ \textsc{Tv-Name}$$

$$\frac{}{\Gamma \vdash \mathbf{0}}\ \textsc{Tv-Nil} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash\, !P}\ \textsc{Tv-Rep}$$

$$\frac{\Gamma \vdash x : (\vec{T}) \quad \Gamma, a_0 : T_0, \ldots, a_i : T_i \vdash P}{\Gamma \vdash x(\vec{a}).P}\ \textsc{Tv-In}$$

$$\frac{\Gamma \vdash x : (\vec{T}) \quad \Gamma \vdash v_i : T_i \quad \forall i \in I}{\Gamma \vdash \overline{x}\langle \vec{v} \rangle}\ \textsc{Tv-Out}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}\ \textsc{Tv-Par} \qquad \frac{\Gamma, x : (\vec{T}) \vdash P}{\Gamma \vdash (\nu\, x : (\vec{T}))\, P}\ \textsc{Tv-Res}$$

Figure 2.7: Typing rules for the π-calculus

2.3.3 Typing Rules

To check if a certain process is well typed we navigate through each element it comprises and we verify two conditions: if the names being used are known and if there are no type mismatches. For this operation we need a type environment; this data structure holds defined names and the associated type. The type environment's syntax is present on Figure 2.4. This is an example of its usage:

$$\varnothing, x : str, y : (str, (str))$$

This type environment contains two type definitions: the name x is defined with type str and the name y is defined with the type $(str, (str))$.

We will now discuss the rules present on Figure 2.7. Rule Tv-Base states that primitive values (strings and numbers) are well typed. Rule Tv-Name sets forth that a name is well typed if it is defined in the type environment and the type used matches the name's declaration. The inactive process $\mathbf{0}$ is always well typed, rule Tv-Nil. Using rule Tv-Res, a restriction, $(\nu\, x : (\vec{T}))\, P$, is well typed if, by adding the association between name of the restriction, x, and the type, $(\vec{T})$, to the type environment, the contained process, P, is well typed. Tv-In rules that the input process, $x(\vec{a}).P$, is well typed if the name

of the input channel, x, is defined with the link type and if, by mapping each name of the input channel's arguments to the corresponding type of x, the contained process, P, is well typed. The output process, $\overline{x}\langle\vec{v}\rangle$, is well typed if its name, x, is declared as link type and if its arguments are correctly typed, rule Tv-Out. The consistency of the replicated process depends on the consistency of the process being replicated, rule Tv-Rep. The parallel process is well typed if each of the composing processes are well typed, rule Tv-Par.

There are two processes that can be used to abstract names: the input and the restriction. The restriction process is used to abstract the number and the type of values a channel can communicate. The input process is used to declare names and the associated types that will be used in the contained process.

As shown in Figure 2.7, our type system ensures the correct usage of values and names on channels. It also constrains us to typify each name we use. Finally, it safeguards the link usage throughout a scope by disallowing arity and type mismatches.

Let us verify if Process 2.3 is well typed:

$$echo \notin \varnothing \Rightarrow \varnothing \nvdash echo(msg, reply).\overline{reply}\langle msg\rangle \quad \text{Tv-In}$$

Since the name *echo* is not present in our typing environment, $\varnothing$, the rule Tv-In fails to typify the process. Hence, in order to correctly typify Process 2.3 we have to declare the name *echo* with the link type:

$$(\nu\, echo\colon (str, (str)))\, echo(msg, reply).\overline{reply}\langle msg\rangle \tag{2.8}$$

The expression $(\nu\, echo\colon (str, (str)))$ can be read as: create a new channel, named *echo*, that communicates a string and a channel. The channel, on the second argument, communicates a string. The name *echo* can only be used in the process held by the restriction process, the held process is Process 2.3.

Now, we show that Process 2.8 is well typed. Using rule Tv-Res we have the derivation

$$\frac{\varnothing, echo\colon (str, (str)) \vdash echo(msg, reply).\overline{reply}\langle msg\rangle}{\varnothing \vdash (\nu\, echo\colon (str, (str)))\, echo(msg, reply).\overline{reply}\langle msg\rangle}\ \text{Tv-Res}$$

Let $\Gamma' \stackrel{\text{def}}{=} \varnothing, echo\colon (str, (str))$. We need to prove that the new typing

environment, Γ', typifies Process 2.3, applying rule Tv-In:

$$\frac{\Gamma', msg\colon str, reply\colon (str) \vdash \overline{reply}\langle msg\rangle \quad \Gamma' \vdash echo\colon (str, (str))}{\Gamma' \vdash echo(msg, reply).\overline{reply}\langle msg\rangle}\ \textsc{Tv-In}$$

Rule Tv-Name ensures that $\Gamma' \vdash echo\colon (str, (str))$ holds. Now, let

$$\Gamma'' \stackrel{\text{def}}{=} \Gamma', msg\colon str, reply\colon (str)$$

We are left with the second sequent, that also holds

$$\frac{\dfrac{}{\Gamma'' \vdash reply\colon (str)}\ \textsc{Tv-Name} \qquad \dfrac{}{\Gamma'' \vdash msg\colon str}\ \textsc{Tv-Name}}{\Gamma'' \vdash \overline{reply}\langle msg\rangle}\ \textsc{Tv-Out}$$

Chapter 3

The Frontend of the Compiler

3.1 Introduction

A compiler is a program that translates a high-level programming language into machine code [11]. The high-level language is also known as source language. Some compilers, however, do not generate machine code, they generate an intermediate language that is translated by another program into machine code.

The compilation process may be divided into three main stages: the syntactic analysis, the semantic analysis, and the code generation. The syntactic analysis consists on the identification of tokens, patterns of characters, on a file. These tokens can be a number, a word, or a more complex format like a string. The semantic analysis stage comprises the validation of the structure of syntactic tokens and the application of rules to the syntactic structure. Code generation synthesises a low-level language from the verified representation. Optimization techniques can be used, in the last stage, to generate faster or smaller code.

A compiler may be separated into two parts: a frontend and a backend. The frontend comprises the syntactical analysis and the semantic analysis. The backend is the code generation part.

The architecture of compilers has been thoroughly studied, so it is possible to develop a compiler in well isolated components. Each component normally represents a compilation stage.

Our compiler's design follows closely [1], covering the chapters from 1 to 5. However, we made some different design choices, still negligible. For treating

symbols and symbol tables, we reuse a package, from the TyCo compiler [21].

3.2 Lexical Analysis

The focus of the lexical analysis is to identify patterns of data in an input stream from the source language. These patterns, called *tokens*, represent different combinations of data, for example, one token can represent a number whilst another one can represent a keyword. The generated tokens are used by the syntactical analysis process. If the input stream contains garbled data, the lexer produces an error and terminates the lexing stage.

The tool used to implement the lexical analysis, was JFlex [9]. This program has a Domain Specific Language (DSL) that is used to generate Java code that implements our lexer.

A JFlex file encompasses three sections: user code, options and declarations, and lexical rules. The user code section contains Java code that is copied verbatim to the generated class. In the options and declaration section we can fine tune the generated class. The lexical rules section is used to map regular expressions to actions. An example of a lexical rule is:

```
59  "0"          { return token(sym.ZERO); }
```

The regular expression that matches the character `0` is mapped to the action that returns a token with the symbol `sym.ZERO`.

The lexical rule to identify the keyword `int` is straightforward:

```
72  "int"        { return token(sym.INT_TYPE); }
```

The following code listing shows the parsing of a name:

```
75  [a-zA-Z][a-zA-Z0-9]*  { return token(sym.ID, yytext());
    }
```

JFlex has the concept of states. Each state defines an independent section of rules. One of the possible actions performed, when an expression is matched, is to change to a different state. States are very useful to declare otherwise complex regular expressions like multi-line comments and strings with escape characters. The following excerpt illustrates the handling of strings:

```
53  <YYINITIAL> {
74  \"           { scanner.startString(); yybegin(STRING); }
```

```
 81 }
 92 <STRING>{
 93 \\\\            { scanner.append("\\"); }
 94 \\\"            { scanner.append("\""); }
 95 \\n             { scanner.append("\n"); }
 96 \\t             { scanner.append("\t"); }
 97 \"              { yybegin(YYINITIAL); return token(sym.
      STRING, scanner.closeString());}
 98 \n              { scanner.append("\n"); }
 99 \r              { }
100 .               { scanner.append(yytext()); }
101 }
```

For the sake of brevity we omit unrelated code. We use an auxiliary data structure to keep the parsed string, the `scanner`. The contents of the string are fed, on line 100, until the scanner finds a double quotes character, on line 97. Special characters can be escaped with the backslash character, on lines 93–96. When the double quotes character is found, the scanner state is changed back to `YYINITIAL` and the token `sym.STRING` is returned, holding the scanned text.

3.3 Syntactical Analysis

The syntactical analysis is the verification of the structure of the input code. During the identification of tokens, the syntactical analysis verifies if they have a valid structure, through the use of a language grammar. The result of the syntactical analysis is an abstract representation of our language, an Abstract Syntactic Tree (AST). The AST is used for latter steps of the compiler, where we need to analysis the meaning of this structure.

The tool used to generate the parser was JavaCup [19]. JavaCup's DSL defines a grammar annotated with Java code, that is used to create the AST. JavaCup's parser copes with multiple syntactic errors, by replacing invalid AST branches with default values and then proceeding with the analysis.

An AST maps the language's grammar. The structure of each class follows conventions that help keep the code predictable and maintainable.

The type hierarchy of the AST is the direct representation of the grammar it implements. Figure 3.1 depicts the hierarchy of π processes that we implemented. Each symbol is a class that extends the root class `Absyn`. Each

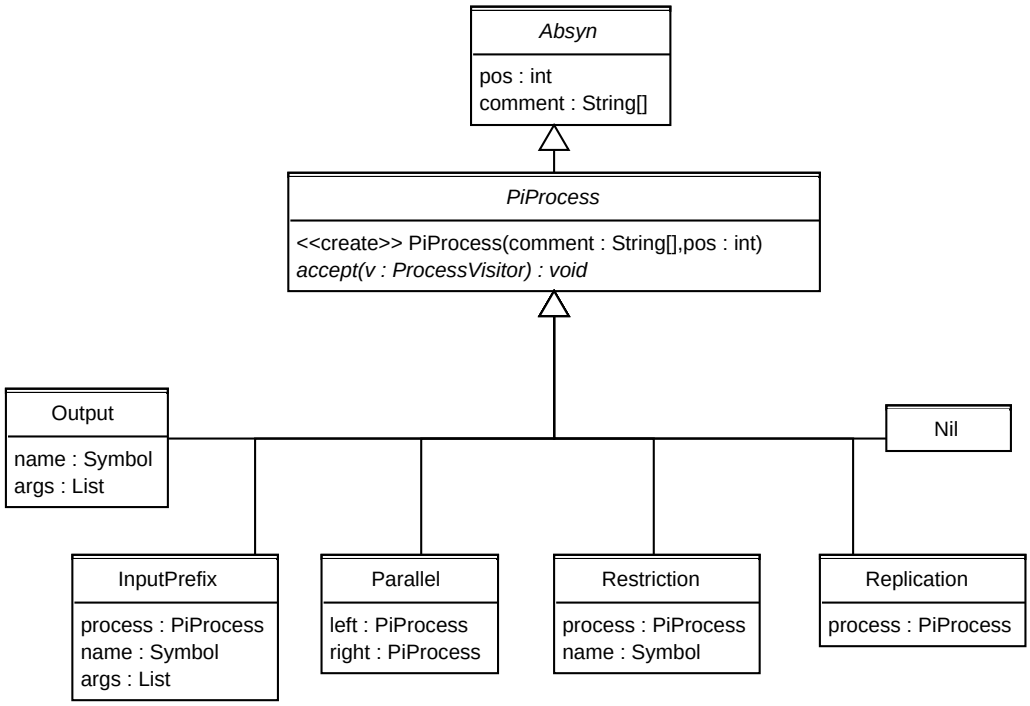

Figure 3.1: Classes related to the symbol process

production corresponds to a class that extends the symbol class. For example the symbol *Proc*, a process, is implemented by the class `PiProcess`. The nil process is the production *Proc* ::= `0`, therefore it must extend the class `PiProcess`.

```
1  public class Nil extends PiProcess {
2    // ...
3  }
```

This is an excerpt of the grammar that generates the inactive process:

```
157 simple_proc ::=
158  ZERO:z {:
159   RESULT = new Nil(new String[0], zleft);
174  :};
```

The symbol `simple_proc` is analogous to the class `PiProcess`. The lexical token `ZERO` is a production of the symbol that represents processes. The reserved variable `RESULT` keeps the value associated with this production.

Each symbol is associated with meta-data related to its contextual location in the source code and in the syntactical structure. These meta-data are implemented by simple class attributes. Each class attribute is affected by the `public` and the `final` modifiers, to pertain their immutable nature. The `Absyn` class, and therefore all symbols, has an attribute that identifies the symbol's offset (its position) in the source code and an attribute that is a user comment. The code listing below is the implementation of the class `Absyn`:

```
 9  public abstract class Absyn {
13      public final int pos;
18      public final java.lang.String [] comment;
28      public Absyn (java.lang.String [] comment, int pos)
            {
29          this.comment = comment;
30          this.pos = pos;
31      }
33  }
```

The implementation is straightforward. The modifier `abstract` enforces the class' nature to be extended only.

The following example shows the implementation of the parallel process: defined by the `Parallel` class. This class is composed by two attributes: the

right process corresponds to the `public final PiProcess right` attribute and the left process corresponds to the `public final PiProcess left`. The code for class `Parallel` is sketched below:

```
10 public class Parallel extends PiProcess {
11     public final PiProcess left;
12     public final PiProcess right;
22     public Parallel(String[] comment, int pos,
           PiProcess left, PiProcess right) {
23         super(comment, pos);
24         this.left = left;
25         this.right = right;
26     }
28     public Parallel(PiProcess left, PiProcess right) {
29         this(new String[0], 0, left, right);
30     }
47 }
```

The string value, one of the basic values ($\vec{v}$), encapsulates a Java string, implemented by the attribute `public final String value`. Its implementation is as follows:

```
 9 public class StringValue extends Value {
14     public final String value;
24     public StringValue(String[] comment, int pos,
           String value) {
25         super(comment, pos);
26         this.value = value;
27     }
35     public StringValue(String value) {
36         this(new String[0], -1, value);
37     }
52 }
```

The remaining classes are implemented in a similar way, following strictly the syntax defined in Figure 2.3

3.4 Static Semantic

Static semantic validity is the application of the typing rules in Figure 2.7 to the AST. The process of applying typing rules is performed by navigating the syntactic tree. Our Compiler uses the Visitor pattern [7] to navigate the AST [1]. We start by presenting this pattern and analyse how the type checking is attained.

3.4.1 The Visitor Pattern

The objective of design patterns is to identify a recurring problem and provide a solution to this problem [7]. Design patterns are catalogued with a comprehensible name that helps the communication of software developers, the description of the problem it targets, the solution proposed to the problem, and the consequences of applying the pattern.

The visitor is a behavioural design pattern, which address the interaction between objects and the assignment of responsibilities. The visitor pattern outlines the application of an operation to a hierarchy of objects, facilitating the addition of new operations without modifying the transversed objects.

This pattern encapsulates behaviour in one object: the visitor. It works by navigating a structure of objects and allowing the user to specify logic code when each element is visited. The visitor has a method named `visit` that accepts an argument, the class of the visited element; this method may be overloaded if there are more than one class of visited elements. The method `visit` implements a dispatching algorithm that maps the type of the element to a method of the visitor.

For example, consider the classes: `Replication` and `Nil`. Both classes share the same superclass, `PiProcess`. This means the method `visit` has the following signature:

```
public void visit(PiProcess element);
```

The name of the methods used for dispatching follows a convention of the concatenation of *case* with the name of the class being dispatched (e.g., Replication). Their signatures are:

```
public void caseReplication(Replication element);
public void caseNil(Nil element);
```

If we call the method `visit` with an instance of the class `Replication`, the

method `caseReplication` is invoked. The situation is the same for instances of class `Nil`.

We have only covered the dispatching, but we mentioned the transversing of a *structure*. Class `Replication` has one attribute of type `PiProcess`:

```
class Replication extends PiProcess {
    public final PiProcess process;
}
```

A visitor class must navigate this structure, so, when the method `visit` is called with an argument of type `Replication`, the method `caseReplicaton` is invoked first and then the method `visit` is called to navigate its attribute.

Each compiler's step entails transversing the AST. The visitor pattern enables the representation of each compilation stage with a visitor class. By using this pattern the compiler becomes modular and extensible. Compilation steps can be added gradually without affecting other packages, because adding new operations does not affect the AST, nor existing classes.

The compiler works on three structure of objects: processes, values, and types. We have created a visitor interface to navigate process trees (`ProcessVisitor`), one to transverse the type trees (`TypeVisitor`), and another for value trees (`ValueVisitor`).

There are various implementations of the Visitor pattern, each of which uses a different dispatching algorithm. We use the Double Dispatch [7] implementation for the Visitor pattern, pointed by [1]. This pattern delegates the dispatching algorithm to each visited element. The Double Dispatch design pattern has a name that suggests its implementation. Every call to a visit will be dispatched by two objects. The first dispatch is from the visitor to the visited object (via dynamic binding). The second dispatch is from the visited back to the visitor (via a method call).

Figure 3.2 illustrates the dispatching algorithm. Each visited element must contain the method `accept`:

```
public void accept(ProcessVisitor v);
```

The method has one argument, the visitor. It is the responsibility of the visited element to call the method with the dispatched code. In the case of the class `Nil`:

```
class Nil extends PiProcess {
    public void accept(Visitor v) {
        v.caseNil(this);
```

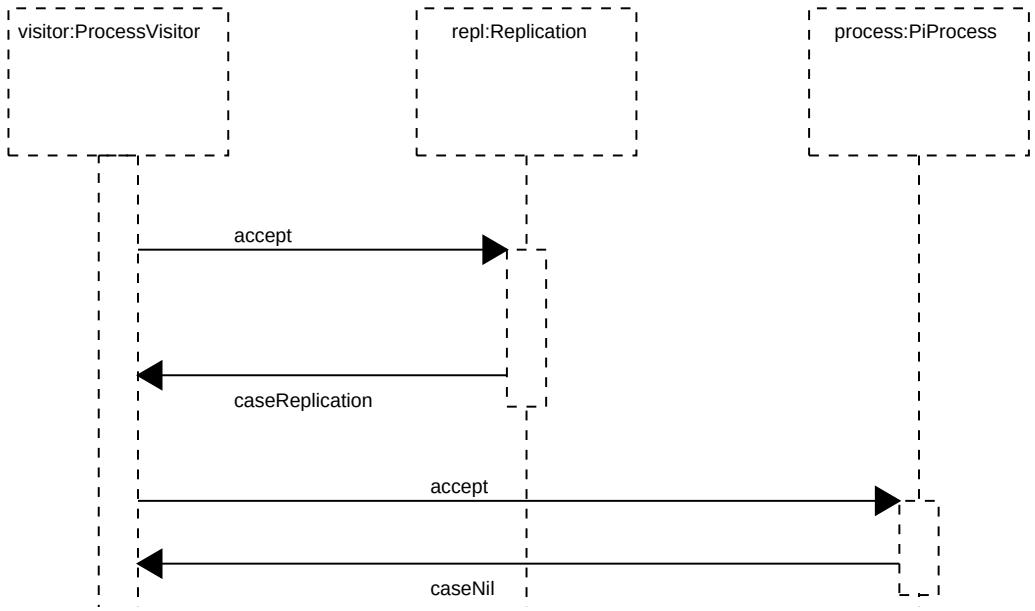

Figure 3.2: Visiting a `Replication` object containing a `Nil` object

```
    }
}
```

The implementation of the method `visit`, of the visitor, is very simple:

```
class DepthVisitor implements ProcessVisitor {
    public void visit(PiProcess element) {
      element.accept(this);
    }
}
```

When the method `visit` is called with an instance of `Nil`, then the method `accept` of class `Nil` is called (the first dispatch). Then, the method `caseNil`, of the class `DepthVisitor`, is called and the user code is executed (the second dispatch).

The analysis of structurally composite objects, like the class `Parallel`, is important to the understanding of where the navigation code is implemented in the Double Dispatch pattern. The navigation algorithm is implemented in the visitor class, not in the visited elements. Hence, the method `accept` of class `Replication` is very similar to the one in class `Nil`:

```
class Replication extends PiProcess {
    public final PiProcess process;
    public void accept(ProcssVisitor v) {
        v.caseReplication(this);
    }
}
```

The implementation of the method `caseReplication` will transverse deeper into the structure:

```
class DepthVisitor implements ProcessVisitor {
    public void visit(PiProcess element) {
        element.accept(this);
    }
    public void caseReplication(Parallel element) {
        visit(element.process);
    }
    // ...
}
```

A triggered event corresponds to the invocation of a dispatched method. The convention for the naming of methods that map events is the name of the event concatenated with the name of the class, for example the event *in* and the class `Output` will be mapped to the method named `inOutput`. Most of the nodes, in this compiler, contain three events: *case*, *in*, and *out*. The *case* event is dispatched by the visitor. The event *in* is triggered after the *case* event, but before transversing deeper into the tree. The *out* event is raised when the visitor leaves the element.

The next code listing shows the how events of class `Parallel` are raised, in class `DepthVisitor`:

```
46  public void caseParallel(Parallel s) throws Exception
        {
48   inParallel(s);
49   visit(s.left);
50   betweenParallel(s);
51   visit(s.right);
52   outParallel(s);
53  }
```

Class `Parallel`, has the event *between*, that is emitted, on line 50, after the

process on the left has been transversed, but before the visitor navigates deeper into the process on the right.

The navigation behaviour has been separated from the event reaction, by using the Decorator pattern, which is a structural design pattern. Structural design patterns focus on the composition of objects to assemble larger or more functional structures. The Decorator pattern extends an operation dynamically, by wrapping an object and delegating existing functionalities to the decorated object. The decorator class, in our compiler, is the `DepthVisitor`. This class transverses the AST and dispatches the events to the wrapped visitor, hence encapsulating the transversing behaviour.

We show the code listing of the method `caseInputPrefix` (of class `DepthVisitor`):

```
94  public void caseInputPrefix(InputPrefix p) throws
        Exception {
95    visitor.caseInputPrefix(p);
96    inInputPrefix(p);
97    visit(p.process);
98    outInputPrefix(p);
100 }
```

The algorithm is depicted in Figure 3.3. The event *case* is used to navigate deeper into the tree and to dispatch the events *in* and *out*, line 96 and line 98. Events are then propagated to the decorated visitor, for example, the method `inInputPrefix` has this implementation

```
101 public void inInputPrefix(InputPrefix p) throws
        Exception {
102   visitor.inInputPrefix(p);
103 }
```

and the `outInputPrefix` has this code

```
104 public void outInputPrefix(InputPrefix p) throws
        Exception {
105   visitor.outInputPrefix(p);
106 }
```

The order in which events are dispatched defines the order in which the tree is transversed. For example, the method `caseInputPrefix` begins with the propagation of the *case* event to the decorated object (line 95). Afterwards, the *in* event is triggered, which, as we have seen already, propagates

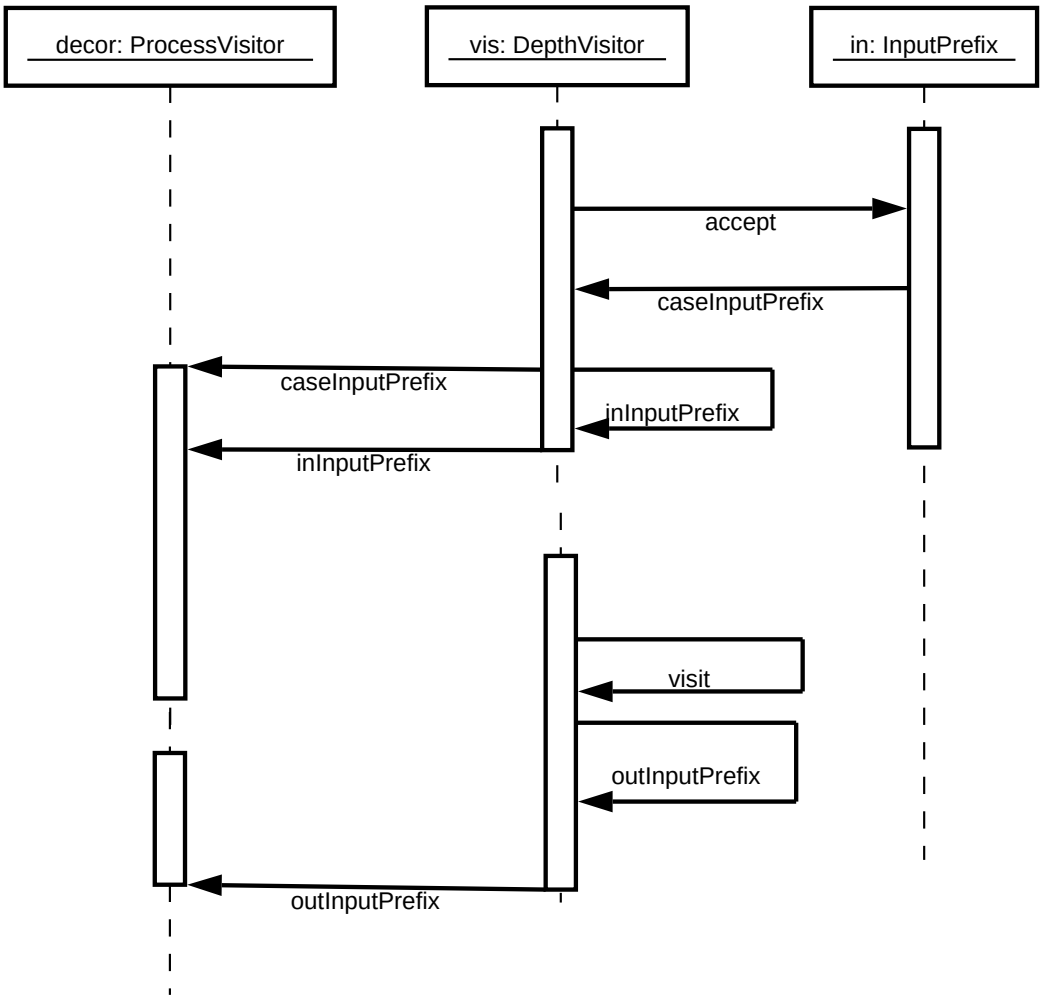

Figure 3.3: The decorated visitor dispatching the `InputPrefix`

the event to the decorated object. Then, the visitor goes deeper into the tree, by transversing the child process

```
97    visit(p.process);
```

Finally the event *out* is dispatched, the event, propagating the event to the wrapped object.

Another example of event dispatching is the handling of the class `Parallel`:

```
46  public void caseParallel(Parallel s) throws Exception
      {
47    visitor.caseParallel(s);
48    inParallel(s);
49    visit(s.left);
50    betweenParallel(s);
51    visit(s.right);
52    outParallel(s);
53  }
54  public void inParallel(Parallel s) throws Exception {
55    visitor.inParallel(s);
56  }
57  public void betweenParallel(Parallel s) throws
      Exception {
58    visitor.betweenParallel(s);
59  }
60  public void outParallel(Parallel s) throws Exception {
61    visitor.outParallel(s);
62  }
```

On line 47, we propagate the event to the decorated object. Afterwards, we emit the event *in*. Then, we go deeper into the tree of processes, by visiting the element on the left. Next, we trigger the event *between* and we visit the element on the right, lines 50–51. Finally, we raise the event *out*.

The structure used on the `InputPrefix` also applies to the class `Parallel`. On each method that implements an event, first we dispatch it to the decorated visitor. In the *case* event we navigate deeper into the contained elements and emit the appropriate events before (or after) visiting each nested element.

3.4.2 Type Checking

We implement the typing rules (Figure 2.7) with the visitor `SemanticChecker`. The transversing of the AST is handled by the `DepthVisitor`, therefore we only need to reason about the application of each rule. Typing rules have a direct representation in the class `SemanticChecker`.

The typing environment is a mapping between symbols and a type definitions. The map, however, lacks the concept of scopes, needed by rules Tv-In and Tv-Res. Thus, the type environment needs to be represented by a symbol table, implemented by class `org.tyco.common.Table`.

Rule Tv-Base and rule Tv-Name infer the type of a value. If the value is a name, we can use the type environment to obtain the type, but if it is a basic value we need another strategy. The class `TypeMapper` uses the type environment and simple comparison of the class of the value to retrieve the type of a value.

The rule Tv-Nil is an axiom. Rules Tv-Par and Tv-Rep are validated as the visitor transverses the AST.

Rule Tv-In is enforced on the following code listing:

```
84  public void inInputPrefix(InputPrefix p) throws
      SemanticException {
85   symbols.beginScope();
88   LinkType lnk = getLink(p, p.name, p.args.size());
89   if (lnk == null) {
90    return;
91   }
94   Iterator<NameValue> iter = p.args.iterator();
95   for(PiType t: lnk.args) {
96    NameValue v = iter.next();
97    symbols.put(v.symbol, t, v.pos);
98   }
104 }
110 public void outInputPrefix(InputPrefix p) {
111  symbols.endScope();
112 }
```

Line 85 creates a new scope, where the arguments of the input channel will be declared (lines 94–98). Next, on lines 88–91, we verify if the name of the channel is declared as a channel type with matching arguments (premise

$\Gamma \vdash x\colon (\vec{T})$). Afterwards, on lines 94–98, we declare the names of the arguments as denoted on the premise $\Gamma, a_0\colon T_0, \ldots, a_i\colon T_i \vdash P$. Finally, method `outInputPrefix` ends the scope as the visitor leaves this element.

Rule Tv-Res asserts the declaration of a name in a process. The code listing below covers the implementation of this rule:

```
145  public void caseRestriction(Restriction s) throws
       Exception {
146   symbols.beginScope();
147
148   if (s.type instanceof LinkType) {
149    symbols.put(s.name, s.type, s.pos);
150   } else {
151    log.restrictionsForChannelsOnly(s, s.name, s.type);
152   }
153  }
159  public void outRestriction(Restriction s) throws
       Exception {
160   symbols.endScope();
161  }
```

When visiting an instance of class `Restriction`, a new scope is created (line 146) and the names are declared (line 149). Then, after the visitor leaves the object the created scope is terminated (line 160). The implementation of rule Tv-Out is similar to rule Tv-In.

3.5 Test Driven Development

The compiler was developed with Test Driven Development (TDD) in mind [3]. TDD bestows a methodology to develop software and it is supported by a simple framework, the *xUnit*. The xUnit is implemented in various languages, on Java it is called JUnit, and, for instance, on Python it is called pyunit.

Unit tests validate a unit of source code, usually a class, providing a written contract that the unit must abide. The xUnit furnishes a framework of common tests and of tools for running tests together, called test suites.

JUnit is part of the Integrated Development Environment (IDE) we used to develop our compiler, the Eclipse IDE. Having an automated and integrated way to test our code helps the adoption of the TDD.

When developing with TDD, the interface is tested before the implementation. The development follows a three step procedure. The first step is to create the test case for the new feature. The second step is to make the test case pass. The third, and final, step is to remove code duplication, by refactoring the code just written.

TDD foments a safe and gradual progression. Providing tests right from the beginning. Each component is built upon tested ones. The developer knows what is working and can concentrate in the next step.

The created tests may serve as additional documentation. Tests are used to show what an object may and may not do, working as examples that highlight the behaviour of objects.

Each step of the compiler's frontend was subject to tests. The tests used on the compiler's development are kept in the package `pi.test`. We used the JUnit library [4], authored by Kent Beck, one of the most important proponents of the TDD. The class `ParserTest` tests the scanner, the parser, and the creation of the AST. The class `PrettyPrinterTest` tests the depth visitor and the pretty printer. The class `OutputTest` tests both the scanner and the pretty printer. The static semantic analysis tests are performed on the class `SemanticCheckerTest`.

The first module to test is the AST, from package `pi.absyn`. We verify the visitor algorithm present in `DepthVisitor`, by testing a simple client class, the `PrettyPrinter`, in the class `PrettyPrinterTest`. This test works by comparing the string generated by the pretty printer with the one we expect. For example, an instance of the class `Nil` must generate the string `'0'`:

```
37  public void testNil() throws Exception {
38   assertProc("Nil␣object", "0", new Nil());
39  }
```

Testing the class `Restriction` is similar:

```
49  public void testRestriction() throws Exception {
50   assertProc("Restriction␣with␣sum", "(new␣a:())(a().0␣
        |␣b().0)",
51     new Restriction("a", new LinkType(new ArrayList<
          PiType>()),
52       new Parallel(new InputPrefix("a", new Nil()), new
            InputPrefix("b", new Nil()))));
```

```
53    assertProc("Restriction␣with␣parallel", "(new␣a:(int)
          )(0␣|␣0)",
54      new Restriction("a", new LinkType(Arrays
55        .asList((PiType) BasicType.INT)), new Parallel(
              new Nil(),
56        new Nil())));
57
58  }
```

To test the parser we need to be able to compare the parsed AST with the expected tree. Each type present in the AST must implement the method `equalsTo`, in order to perform the comparison of trees. The parser tests are implemented in the class `ParserTest`. We have implemented a helper factory of the parser that generates the AST from a string. The tests cover combinations of processes, of values, and of types. This code listing shows the tests performed on class `Parallel`:

```
92  public void testParallel() throws Exception {
93    assertCode("0␣|␣0", new Parallel(new Nil(), new Nil()
          ));
94  }
```

Constructing the AST by hand is a tedious task. This is an argument promoting the creation of less tests. In order to make the testing process easier, we created tests comparing a source code to the generated pretty printed code in the class `OutputTest`. The source code, provided as a string, is parsed and an AST is generated. Next, this AST is pretty printed and the provided string is compared with the generated string. Finally, the pretty printed string is parsed once more and the two AST's are compared. On this stage three verifications are performed and the tests are easier to create than the previous two tests. However, this test is only possible when all the parts that compose it are already well tested. As an example we list the code showing the testing on arithmetic operations:

```
117  public void testAritmetics() throws Exception {
118    assertCode("a<1␣*␣2␣+␣3>");
119    assertCode("a<1␣+␣2␣*␣3>");
120    assertCode("a<−1>");
121    assertCode("a<1␣*␣−1>");
122    assertCode("a<1␣/␣−1>");
```

```
123     assertCode("a<1 + -1>");
124     assertCode("a<-1 % -1>");
125     assertCode("parenthisis", "a<(1)>", "a<1>");
126     assertCode("parenthisis", "a<(1 + 1)>", "a<1 + 1>");
127     assertCode("parenthisis", "a<(1 * 1)>", "a<1 * 1>");
128     assertCode("parenthisis", "a<(-1)>", "a<-1>");
129     assertCode("parenthisis", "a<(1) + (1)>", "a<1 + 1>")
            ;
130     assertCode("parenthisis", "a<(1) % (1)>", "a<1 % 1>")
            ;
131   }
```

The tests covering the semantic analysis, in class `SemanticCheckerTest`, validate the implementation of the typing rules. In these tests we can create erroneous code and verify if the errors raised by the parser (or not) are the ones we expect. We can also create valid code and verify how the parser copes with it:

```
61   public void testChecker() throws Exception {
62     assertInvalid("a().0", ErrorCode.UNDEF_SYMBOL); // a
           is undifined
63     assertValid("(new a:())a().0");
64
65     assertInvalid("z is undifined", "(new a:(int))a<z+1>
           ", ErrorCode.UNDEF_SYMBOL); // z is undifined
66     assertInvalid("(new a:(int))(new b:(str))b(z).a<z+1>
           ", ErrorCode.TYPE_MISMATCH); // z should be an
           integer
67     assertInvalid("a is defined on other scope", "a(b).0
           | (new a:()) 0", ErrorCode.UNDEF_SYMBOL);
68     assertInvalid("((new a:()) 0) | a(b).0", ErrorCode.
           UNDEF_SYMBOL);
69     assertInvalid("restriction applies to nil and not to
           parallel", "(new a:()) 0 | a(b).0", ErrorCode.
           UNDEF_SYMBOL);
70     assertValid("(new a:())(a().0 | (new a:(int))a(b).0)"
           );
71     assertInvalid("(new a:int)0", ErrorCode.
           SYNTATIC_ERROR, ErrorCode.SYNTATIC_ERROR);
```

```
72    }
```

The TDD was useful to stress the correctness of the various parts of the frontend. Tests do not ensure that the code is correct, but they do show what conditions have been considered. Automated tests also help maintain the quality of the code because regressions are detected quicker.

Chapter 4

Conclusion and Further Work

4.1 Conclusion

We presented the π-calculus along with a frontend of a compiler for a language based on this calculus. Our study of the π-calculus, compulsorily, fell upon computer science topics like mobile computing, process algebra [2], and concurrency. The study on the π-calculus was also backed up by other topics, like the microprocessors' future [16] and the theory of types [5].

By following Andrew Appel's [1] design and simple conventions we were able to construct a modular and maintainable compiler's frontend. The Visitor design pattern imposed the modular design we achieved and defined the compiler's architecture.

The development of the compiler was guided by important software engineering techniques like Test Driven Development, Design Patterns, and Refactoring. TDD helped validate the work done. Refactoring made the code cleaner, with less redundancy. Design patterns helped with solutions to known problems.

4.2 Further Work

The compiler is left unfinished, the backend is missing. In this section we show a possible refactoring to the visitor we have implemented and introduce the backend of the compiler.

4.2.1 Refactoring the Visitor

The Double Dispatch pattern is hindered by a cross cutting concern [8]. The dispatching functionality is scattered along the depth visitor and along each visited element. The functionality present in each visited elements contains code that it is highly repetitive and error prone, hinting us to code that may need refactoring.

Concerning code redundancy first, each visited element contains a method named `accept` that calls a visitor's method following the convention we explained before. Following is the dispatching code present on the Nil class:

```
37  public void accept(ProcessVisitor v) throws Exception
        {
38   v.caseNil(this);
39  }
```

The Java reflection Application Programming Interface (API) allows dynamic introspection of an object. With this API we can obtain properties of a class, such as which methods it comprises. It also enables invoking a method by its name. Introspection is useful for automating repetitive code that follows a convention.

Using introspection, the method can be implemented as:

```
Method meth = v.getClass().getMethod("caseNil", Nil.
    class);
meth.invoke(v, this);
```

The visitor's method `caseNil` is obtained dynamically, then it is invoked, passing the same arguments as before. Note that the variable `v` is now passed explicitly when the method `caseNil` is called.

The next step is to perform the refactoring pattern Pull Up [6]. That refactoring moves repeating code existing on classes that share the same superclass to the superclass. To do so, we abstract the name of the class and the class object passed to the method `getMethod`:

```
String cls_name = this.getClass().getSimpleName();
Method meth = v.getClass().getMethod("case" + cls_name,
     this.getClass());
meth.invoke(v, this);
```

Now the code is ready to be moved to the class `PiProcess`.

The use of introspection slows down the execution, but optimization techniques may be used regain performance. One optimization technique could be achieved by the application of the Flyweight pattern [7] to cache the retrieval of the `Method` object.

Even after refactoring, the dispatching algorithm is scattered between `PiProcess` and `DepthVisitor` classes. The method `accept`, of class `PiProcess`, is generic enough to be moved outside its holding class, because it does not depend on any private attribute. If we move the dispatching algorithm to the class `DepthVisitor`, we remove the cross cutting concern.

A Visitor that uses reflection (introspection) to the dispatching mechanism and is decoupled from the visited classes is called a Reflective Visitor [10]. This kind of Visitor removes the dependency from the type hierarchy and facilitates adding new operations.

4.2.2 The Backend of the Compiler

Another task we want to address in the forthcoming future is the creation of the backend of the compiler. Our compiler is going to target a multi-threaded typed assembly language, MIL [22]. We are going to study the MIL language and code generation algorithms. It will also be of relevant importance of the study concurrency theory, particularly locks algorithms, and the study of typed assembly languages.

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd edition.* Cambridge University Press, 2002.

[2] Jos C. Baeten. A brief history of process algebra. *Theoritical Computer Science*, 335(2-3):131–146, 2005.

[3] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002.

[4] Kent Beck and Erich Gamma. JUnit. http://www.junit.org/.

[5] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

[6] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[9] Gerwin Klein. JFlex. http://jflex.sourceforge.net/.

[10] Yun Mai and Michel de Champlain. A pattern language to visitors. *8th Conference on Pattern Languages of Programs*, 2001.

[11] Ronald Mark. *Writing Compilers and Interpreters*. 1996, Wiley.

[12] Francisco Martins. *Controlling Security Policies in a Distributed Environment*. PhD thesis, Faculty of Sciences, University of Lisbon, 2005.

[13] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[14] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[15] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992.

[16] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[17] Joachim Parrow. An introduction to the pi-calculus. In Jan Bergstra, Alban Ponse, and Scott Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.

[18] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[19] TUM team. JavaCup. http://ww2.cs.tum.edu/projects/cup/.

[20] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, 2-42, pages 230–265, 1936.

[21] Vasco T. Vasconcelos. Core-TyCO, appendix to the language definition, yielding version 0.2. DI/FCUL TR 01–5, Department of Informatics, Faculty of Sciences, University of Lisbon, 2001.

[22] Vasco T. Vasconcelos and Francisco Martins. A multithreaded typed assembly language. In *Proceedings of TV06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, 2006.