Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

POLY Language Specification

Hugo Torres Vieira

Licenciatura em Engenharia Informática
Diploma thesis

Supervisor:
Professor Luís Caires

October 2002

Abstract

This document is a reference manual of the `POLY` programming language, covering it's syntax, type system and semantics.

Contents

List of Figures

Chapter 1

Introduction

The programming paradigms present in the `POLY` language, such as concurrency, mobility and object orientation, illustrate the fact that the study of the language is of very up to date interest. Aiming at such different scopes as general purpose programming, fast prototyping of distributed applications, coordination of software systems or logic agent-based programming, it is clear that it is in fact a very interesting language to develop and study.

Based on Lpi [Cai99], a minimal calculus related to Milner's pi-calculus [Mil99], the language is composed by a minimal set of constructions. These basic elements of the language are persistent and non persistent software agents, messages, restriction of names, parallel composition of processes and the inaction agent.

For a more comprehensive study of all the language details, this report, that can be referred to as the language's reference manual, wrote using as reference [MT91, MTH91], gives a complete definition of the language, firstly by motivating this study through a simple example, then by illustrating the concrete syntax specification, followed by a detailed insight of the type system and finally through a brief explanation of the semantics, finishing off each major chapter with some issues that surfaced during the implementation of the language's static checker.

Chapter 2

Motivation

The better way of introducing the POLY language is to give a simple notion of the constructions of the language. The basic computational element of the language is a software agent, that can diverge into two different sorts: persistent and non-persistent, corresponding to definitions and commands, respectively, that will be more precisely defined further along. The specification of concurrent agents, that are also a key feature of the language, is ensured by the parallel composition construction.

Concurrent agents will certainly require some form of communication platform which is provided by the message construction, that can be seen as a specification of a named channel that holds determined information. The language also supports a restriction construction which corresponds to a scoped fresh name declaration. Finally the inaction construction represents the inactive agent.

Figure 2.1 provides a listing of the referred POLY constructions through a simple grammar, that, in spite of the intended incomplete explanation, provides a sufficient introduction to the basics of the language.

$\langle P,\ Q\rangle$	::=	**inaction**	inaction
	|	$\langle P\rangle$ | $\langle Q\rangle$	composition
	|	**new** $\langle Vars\rangle$ **in** $\langle P\rangle$	restriction
	|	m	message
	|	**def** $< \langle TypeVars\rangle > \langle Vars\rangle$ **in**	
		$\langle Pattern\rangle$ [$\langle P\rangle$] $\langle Q\rangle$	definition
	|	**com** $< \langle TypeVars\rangle > \langle Vars\rangle$ **in**	
		$\langle Pattern\rangle$ [$\langle P\rangle$] $\langle Q\rangle$	command

Figure 2.1: POLY constructions listing

```
def left,right,res:list[integer] in collect(left,right)
  [concatenation(left,right,res)] print(res)
|
collect(cons(1,cons(2,nil)), cons(3,nil))
```

Figure 2.2: A first POLY example

2.1 A POLY program example

In order to give an intuitive notion of the language this section starts by presenting an example that provides a top level view of the language. Figure 2.2 shows a program used to concatenate two integer lists.

Assuming the existence of determined specifications that are not presented, such as the one required for the concatenation algorithm, this program would concatenate two lists, being the result, a list containing three integers (1, 2 and 3), printed in the screen.

The program starts by launching the definition and the message concurrently. If the message is able to match the pattern of the definition, located after the **in** keyword, and if the test is successful, meaning that the process contained in the square brackets successfully reduces (more on this later), the definition is activated.

The activation of the definition will imply the instantiation of the left and right variables with the lists contained in the message, specified through a data constructor that resembles the usual list constructor. Also as a result of the activation of the definition comes the removal of the message from the environment and the instantiation of variable res with the value that makes the test successful, which is the concatenation resulting list. Finally, the last consequence of the definition's activation is the launching of the print message, which is assumed to be collected by a special procedure that prints the list contained in the message to the screen.

2.1.1 A detailed study of the example

The example appears in a complemented version in Figure 2.3, which will help to provide a reference to the study of a number of the language's details.

A POLY program is composed of a sequence of top level declarations and an agent definition, also referred to as a process, being the latter composed by the constructions illustrated previously.

Included in the language top level declarations is the **declare** construction that functions as a name declaration. The name (for example cons) is provided after the **declare** keyword and following the colon comes the associated type as illustrated in (2.1).

```
declare cons:(integer, list[integer])list[integer];
declare concatenation:(list[integer], list[integer],
                       list[integer])msg;

// concatenation(argument list, argument list, result)
def left,right,res:list[integer] in
  concatenation(left,right,res) []
new conc:(list[integer],list[integer])msg in
(
  conc(left,res)
  | def x: integer; lst,r:list[integer] in
      conc(cons(x,lst),cons(x,r)) [] conc(lst,r)
  | com conc(nil,right) [] inaction
)
|
new collect:(list[integer],list[integer])msg in
(
  def left,right,res:list[integer] in collect(left,right)
    [concatenation(left,right,res)] print(res)
  |
  collect(cons(1,cons(2,nil)), cons(3,nil))
)
```

Figure 2.3: A simple POLY program.

$$\underbrace{\textbf{declare}}_{keyword}\ \underbrace{\text{cons}}_{identifier}\ :\underbrace{(\textbf{integer},\ \textbf{list}\,[\textbf{integer}])\,\textbf{list}\,[\textbf{integer}]}_{type};\qquad (2.1)$$

In the example cons is declared as a data constructor which forms a new list of elements placing an element at the head of a list, being the head element specified in the first argument position, the tail list as the second and the resulting list as the return type.

The other declaration present in the example specifies concatenation as being a message, a predefined type that stands for the referred named channels, used for communication purposes. In this particular case the message contains a set of three lists, meaning that when published - when used as a message - the concatenation construct will reside in some sort of execution environment, being it's access ensured by the declared name and it's contents three integer lists.

Following the top level declarations comes the agent specification, that will be more successfully introduced if explained through the evolution of the program. For starters, at the agent top level there is a parallel composition between a definition and a restriction, which means at the first stage a persistent agent and a restriction will be introduced concurrently in the environment.

A definition is composed by declarations, following which comes the **in**

keyword, an acceptance pattern, a test agent - encapsulated by the square brackets - and a continuation agent as shown in (2.2). To be activated two conditions must be validated: there must exist messages in the environment that match the acceptance pattern and the test agent must successfully reduce.

$$\underbrace{\textbf{def}}_{keyword}\ \underbrace{\underbrace{\text{left , right , res}}_{identifiers} : \underbrace{\textbf{list}\,[\textbf{integer}]}_{type}}_{variable\ declaration}\ \underbrace{\textbf{in}}_{keyword}\ \underbrace{\text{collect (left , right)}}_{pattern}$$
$$[\ \underbrace{\text{concatenation(left , right , res)}}_{test\ agent}\ ]\quad \underbrace{\text{print (res)}}_{continuation\ agent} \tag{2.2}$$

In the first definition of the program present in Figure 2.3 a non empty acceptance pattern is present so, to be activated, this definition will require the existence of messages in the execution environment that match it's pattern.

An intuitive idea of the functionality of the acceptance pattern is that it provides a form of specifying an input to the agent, through the placement of information in the environment and consequent recovery through the pattern, being the information removed from the environment after a successful match. The test agent holds the other condition for the activation of the definition and one can say that it functions similarly to a guard that ensures the validity of determined conditions.

The activation of an agent works in a all or nothing fashion: only when both conditions are valid the activation takes place, implicating the removal of the messages that matched the pattern, the instantiation of variables declared in the definition and the publishing of the continuation agent. The variable instantiation occurs due to the matching procedure and to the reduction of the test agent, being the values used in the first case the ones present in the messages that matched the pattern, while in the second case the values correspond to the ones that validate the condition stated by the test.

Assuming, as expected, an empty execution environment at the starting point, the first definition will be impossible to activate due to the lack of messages in the environment. So the evolution of the program must involve the restriction, which functionality is to introduce a fresh name in the environment, regarding that it's scope is the agent present in the body of the restriction.

Bearing in mind that `POLY` programs are capable of concurrent execution these unique names provide an important role in ensuring that agents run in isolation, meaning they to not interfere with each other. In this case this unique name ensures no interference in this simple communication: the published message can only be collected by this definition because the name is known only to these elements present in the body of the restriction.

```
def left , right , res : list [integer] in FRESH_COLLECT( left , right )
  [concatenation( left , right , res )] print( res )
|
FRESH_COLLECT( cons(1 , cons(2 , nil )) , cons(3 , nil ))
```

Figure 2.4: Results of evaluating the restriction.

Focusing only in the body of the restriction after the introduction of a fresh name, one can conceive that the environment is similar to the one shown in Figure 2.4. In this Figure, other than the fresh name, the environment resembles the one presented in the previous section (Figure 2.2), so a shortcut is made directly to the test agent present in the definition, recalling that variables left and right are instantiated with the lists contained in the shown message, because the matching is successful, and res is not yet instantiated.

The condition stated in the test agent illustrated in (2.3) is that the concatenation of two integer lists results in another integer list. For the first and second list determined values are used, specified through the data constructor cons, while a non instantiated variable is used for the third argument, meaning that, if successful, the test reduction will instantiate the variable with the value corresponding to the result of the concatenation.

$$\underbrace{\begin{array}{l}\textit{// concatenation(argument list, argument list, result)}\\ [\,\text{concatenation}(\underbrace{\text{cons}(1,\text{cons}(2,\textbf{nil}))}_{\textit{data constructor}}\,,\ \underbrace{\text{cons}(3,\textbf{nil})}_{\textit{data constructor}}\ ,\ \underbrace{\text{res}}_{\textit{variable}}\ )\,]\end{array}}_{\textit{test agent}} \tag{2.3}$$

To reduce the agent present in the illustrated test, the message construction that composes it is launched in an encapsulated environment, meaning that no interference is made to the initial environment through the reduction of the test agent, except possibly in the instantiation of variables declared in the definition, which makes sense, keeping in mind that a test condition is being handled. If through a series of reductions this encapsulated environment is led to contain only the inaction agent or persistent agents the test is successful.

For the sake of illustration consider that the encapsulated environment referred is at this starting point composed as illustrated in Figure 2.5, regarding that the definition present in the initial environment is accessible because definitions are accessible in all contexts.

One can easily note that the message is "matchable" to the definition's acceptance pattern. Since the definition's test agent is empty and the acceptance pattern has only one term the definition will be activated, causing the instantiation of variables left , right and res with the values contained in the

```
def left,right,res:list[integer] in
  concatenation(left,right,res) []
new conc:(list[integer],list[integer])msg in
(
  conc(left,res)
  | def x: integer; lst,r:list[integer] in
      conc(cons(x,lst),cons(x,r)) [] conc(lst,r)
  | com conc(nil,right) [] inaction
)
|
concatenation(cons(1,cons(2,nil)),cons(3,nil), res)
```

Figure 2.5: The first stage of the test agent reduction.

```
new conc:(list[integer],list[integer])msg in
(
  conc(cons(1,cons(2,nil)),res)
  | def x: integer; l,r:list[integer] in
      conc(cons(x,l),cons(x,r)) [] conc(l,r)
  | com conc(nil,cons(3,nil)) [] inaction
)
```

Figure 2.6: Second stage.

message (cons(1,cons(2,**nil**)), cons(3,**nil**) and res, respectively). Figure 2.6 illustrates the new environment to consider after the activation of the definition, focusing only in the continuation agent, regarding the referred instantiation of variables.

At this stage there is a restriction construction to handle, which will cause the introduction of fresh names in the environment. To help to form an intuitive idea, Figure 2.7 shows the evolution of the program. This is a good tutorial example to explain why these unique names are required to guarantee isolation, because the different executions of the concatenation algorithm might be running concurrently so if no isolation was ensured the necessary message passing to the algorithm would not be protected, and the confusion that would arise from wrong message passing would cause incorrect results.

Following the introduction of the fresh name, the environment is com-

```
FRESH_CONC(cons(1,cons(2,nil)),res)
| def x: integer;l,r:list[integer] in
    FRESH_CONC(cons(x,l),cons(x,r)) [] FRESH_CONC(l,r)
| com FRESH_CONC(nil,cons(3,nil)) [] inaction
```

Figure 2.7: Third stage.

```
FRESH_CONC(cons(2, nil), R)
| def x: integer; lst, r:list[integer] in
    FRESH_CONC(cons(x, lst), cons(x, r)) [] FRESH_CONC(lst, r)
| com FRESH_CONC(nil, cons(3, nil)) [] inaction
```

Figure 2.8: Fourth stage.

```
FRESH_CONC(nil, R')
| def x: integer; lst, r:list[integer] in
    FRESH_CONC(cons(x, lst), cons(x, r)) [] FRESH_CONC(lst, r)
| com FRESH_CONC(nil, cons(3, nil)) [] inaction
```

Figure 2.9: Fifth stage.

posed of a message, a definition and a command. A command is a very similar construction to the definition, differing only in the fact that after activated, the definition remains accessible for usage, while the command does not, thus the persistent and non persistent characterization.

Looking at the command one can realize that the message present in the environment is not "matchable" to the command's acceptance pattern, because the first argument of the message is the list containing elements 1 and 2 (cons(1,cons(2,**nil**))) and would have to be the empty list (**nil**) in order for the matching to succeed. On the other hand the definition's pattern is able to match the message because the first argument assumes the same form, while the second involves an unification of the variable res. Due to the matching x now takes value 1, while lst takes value cons(2,**nil**) and finally res takes the form of cons(1,R) (bearing in mind that x has been instantiated) where R has not an established value. Figure 2.8 shows the resulting environment, differing only in the presented message.

The same procedure would again occur, resulting in the environment shown in Figure 2.9, causing R to take the cons(2,R') form and therefore causing res to take the cons(1, cons(2, R')) form.

The message will now match the command's acceptance pattern, because it's first argument is **nil**, causing R' to be instantiated with cons(3,**nil**) and consequently res will be instantiated with the value cons(1,cons(2,cons(3,**nil**))). As the result of the command activation the inaction agent is published and the command disappears so the resulting environment is as shown in Figure 2.10, composed only by the inaction agent and a definition, which means that the test has succeeded.

Returning to the initial environment, present in Figure 2.4, the success of the test implicates the activation of the definition and therefore the res variable is instantiated with the referred value, which is published through the print construct, a special message that would be collected by a system agent that prints it's contents to the screen.

```
def x: integer; lst,r:list[integer] in
  FRESH_CONC(cons(x,lst),cons(x,r)) [] FRESH_CONC(lst,r)
| inaction
```

Figure 2.10: Sixth stage.

To finish off this introduction, a fundamental element of the language has to be mentioned: types. As expected the basic types are present in the POLY language, namely **integer**, **real**, **string** and **boolean**. To join the set of predefined types comes the very useful **list** type and also the previously referred **msg** type. The type used for the concatenation construct is again shown in (2.4).

$$(\textbf{list}\,[\textbf{integer}],\ \textbf{list}\,[\textbf{integer}],\textbf{list}\,[\textbf{integer}])\textbf{msg} \tag{2.4}$$

In this example the most important type constructions are present, starting at the concatenation type itself, which corresponds to a compound type, and ending at the basic type level (**integer**), not forgetting the parametric type present in the compound type arguments (**list** [**integer**]).

The usual basic types are used as could be expected, while **msg** has a special meaning, standing for a message construction and therefore being capable of placement in the execution environment, and **list** has a usage detail, regarding it's parametric form, which is the specification of the type of the list's elements, being it's generic form submissible to any type instantiation and therefore capable of representing any kind of lists.

The remaining type construction, the compound type, represents a compound structure used for data construction, for example the concatenation construct or the list constructor (cons) type shown in (2.5).

$$(\textbf{integer},\ \textbf{list}\,[\textbf{integer}])\textbf{list}\,[\textbf{integer}] \tag{2.5}$$

Looking at the cons type and keeping in mind that the language supports parametric types, surfaces an idea in order to make the POLY constructors more flexible and useful: polymorphic constructors. The idea present is no surprise, because the **list** type has already introduced it, and it involves the creation of constructors that handle any kind of type. This idea is very useful in the list constructor because the construction involved is completely independent of the type of the elements of the list. The specification of the cons construct as a polymorphic constructor is shown in (2.6).

$$\textbf{declare}\ \text{cons:}\ \underbrace{[\text{X}]}_{\textit{parameter list}}\ (\text{X},\ \textbf{list}\,[\text{X}])\textbf{list}\,[\text{X}]; \tag{2.6}$$

The X identifier represents a type parameter, so when cons is actually used this parameter will be replaced with the actual type that is being used,

being this instantiation done implicitly.

A key feature of the POLY language is the possibility to specify a polymorphic acceptance pattern, allowing, for instance, type generic algorithms to be written as so, instead of being replicated for each type of elements they handle. To introduce these polymorphic patterns one must mention an additional element of the definition (or command), which are the type variables identified in (2.7).

$$\textbf{def} \quad \underbrace{\text{<X>}}_{\textit{type variables}} \quad \text{left , right , res : } \textbf{list}\,[\text{X}]\ \textbf{in}\ \text{concatenation(left , right , res)} \quad [] \ldots \tag{2.7}$$

Identifier X, a type variable declared in the definition, will stand as a universal type, being it's scope the whole agent, from the variable declaration to the continuation agent. The basic idea is that the acceptance pattern, due to the variable's declaration, is receptive to concatenation messages that take any kind of lists as arguments (**list** [X]). This characteristic gives the intended support for generic algorithms, as is, for instance, the concatenation algorithm that works for any kind of lists.

The type variables will work somewhat like value variables, being also instantiated at some point. For type variables the instantiation will occur as a consequence of the matching procedure, being the "values" the types present in the messages. After the pattern matching procedure, a type variable can be seen as a regular type because the referred instantiation will indeed replace it with a regular type.

As one can notice, in POLY, some types can acquire great complexity, and to handle that, it is possible to declare type abbreviations in a construct similar to the previously presented **declare** construct. The specification of this top level name declaration is shown in (2.8) as well as it's usage, using the cons type as the example.

$$\begin{array}{l} \underbrace{\textbf{type}}_{\textit{keyword}}\ \underbrace{\text{consT}}_{\textit{identifier}}\ \underbrace{[\text{X}]}_{\textit{parameter list}}\ : \underbrace{(\text{X},\ \textbf{list}\,[\text{X}])\,\textbf{list}\,[\text{X}]}_{\textit{abbreviated type}}\ ; \\ \textbf{declare}\ \text{cons[W]: consT[W];} \end{array} \tag{2.8}$$

In chapter 4 there is a complete presentation of the type system, which along with the syntax and semantics explanation will give a concise definition of the language, being the information presented at this point sufficient to give a starting point for the language study.

Chapter 3

Syntax

In this chapter the `POLY` language syntax is thoroughly defined through the language's lexical elements and grammar. Following the complete description of the language's syntax, this text provides a simple example of a `POLY` program, aiming at a complementary presentation of the language, and finally focuses on the implementation issues that seem more relevant to mention.

3.1 Lexical Elements

As usual, lexical elements are the basic symbols from which programs are built. Lexical elements are either *literals*, *reserved words*, *special characters*, or *identifiers*. *Literals* denote either special values or types; namely *numeric literals*, *string literals*, *list literals*, *boolean literals*, and *type literals*, where the first are composed of *integer literals* and *real literals*.

$$
\begin{array}{lcl}
\langle \mathit{Literal} \rangle & ::= & \langle \mathit{ValueLiteral} \rangle \mid \langle \mathit{TypeL} \rangle \\
\langle \mathit{ValueLiteral} \rangle & ::= & \langle \mathit{NumericL} \rangle \mid \langle \mathit{StringL} \rangle \mid \langle \mathit{BooleanL} \rangle \mid \langle \mathit{ListL} \rangle \\
\langle \mathit{NumericL} \rangle & ::= & \langle \mathit{IntegerL} \rangle \mid \langle \mathit{RealL} \rangle
\end{array}
$$

The following subsections give a precise definition of all lexical elements.

3.1.1 Numeric literals

Numeric literals are either *integer literals* or *real literals*. An integer literal is a non-empty sequence of digits, eventually preceded by the character "–".

$$
\begin{array}{lcl}
\langle \mathit{Digit} \rangle & ::= & 0 \mid \cdots \mid 9 \\
\langle \mathit{IntegerL} \rangle & ::= & [-] \, \langle \mathit{Digit} \rangle^{+}
\end{array}
$$

A *real literal* is an integer literal possibly followed by the character ".", followed by a sequence of zero or more digits, possibly followed by a "E" character and another integer literal, and where either the "." or the "E" must necessarily occur.

⟨*RealL*⟩ ::= ⟨*IntegerL*⟩ [. ⟨*Digit*⟩$^+$] [E ⟨*IntegerL*⟩]

Examples ⟨*IntegerL*⟩: 12, −12. ⟨*RealL*⟩: −10.0E1, 12E−3, 0.02.

3.1.2 String literals

A string literal is a sequence of character specifications inside double quote characters "". A *character specification* is either

- Any printable UNICODE character.
- The two character sequence "\n", interpreted as the end-of-line character.
- The two character sequence "\t", interpreted as the tabbing character.
- A sequence of characters of the form "*ddd*" where *ddd* is a sequence of up to three digits, interpreted as the UNICODE character with decimal code *ddd*, meaning that *ddd* must be a value less than 256.
- A sequence of characters "\^*a*", where *a* is any character with UNICODE code n in the range $64-95$, interpreted as the UNICODE character with decimal code $n-64$.
- The two character sequence "\"", interpreted as the double quote character "".
- The character "\", causing the contents of the input stream up to the next occurrence of "\" to be ignored, while composed of formatting characters (e.g. "\t", "\n" or " ").

Examples ⟨*StringL*⟩: "\thello\n", "hello\ \ my \" world \"".

3.1.3 Boolean Literals

A boolean literal is either one of the usual boolean constants.

⟨*BooleanL*⟩ ::= **false** | **true**

3.1.4 List Literals

The only list literal is the identifier standing for the empty list.

⟨*ListL*⟩ ::= **nil**

3.1.5 Type Literals

A type literal is either one of the expected basic types, the list type or the message type.

⟨*TypeL*⟩ ::= **integer** | **real** | **string** | **boolean** | **list** | **msg**

3.1.6 Reserved Words

Reserved words look like identifiers, but are actually keywords which signal the constructs of the language. All the reserved words are shown bellow.

⟨*Reserved*⟩ ::= **com** | **declare** | **def** | **in** | **inaction** | **infix** | **new** | **prefix** | **suffix** | **type**

3.1.7 Identifiers

An *identifier* is a sequence of characters, chosen either among the upper and lower case letters, the digits, and the underscore character, or among the special *symbol characters*, in such a way that the first character is either a letter or a symbol character. In the first case, the identifier is called a *name*, in the second case the identifier is called an *operator*.

⟨*Upper*⟩ ::= A | ··· | Z
⟨*Lower*⟩ ::= a | ··· | z
⟨*Symbol*⟩ ::= / | ! | $ | # | % | & | ∗ | + | . | : | < | > | = | ? | @ | ^ | | | ~ | − | \
⟨*Letter*⟩ ::= ⟨*Lower*⟩ | ⟨*Upper*⟩ | _
⟨*Identifier*⟩ ::= ⟨*Operator*⟩ | ⟨*Name*⟩
⟨*Operator*⟩ ::= ⟨*Symbol*⟩$^+$
⟨*Name*⟩ ::= ⟨*Letter*⟩ (⟨*Letter*⟩ | ⟨*Digit*⟩)*

Examples ⟨*Name*⟩: Hello2, poly_lang. ⟨*Operator*⟩: $$, +.

3.1.8 Special Characters

The following characters have reserved meanings.

⟨*SpecialChar*⟩ ::= [|] | : | ; | | | & | (|) | , | < | >

3.1.9 Comments

Any text between "/∗" and "∗/", and any text from an occurrence of "//" to the end of the line, is considered a comment in the source code.

3.2 Grammar

3.2.1 Compilation Units

A POLY application is a *compilation unit*. A compilation unit is composed by a series of type and symbol declarations, possibly followed by a process (the body of the compilation unit).

⟨*Unit*⟩	::=	⟨*Declarations*⟩ [⟨*Process*⟩]
⟨*Declarations*⟩	::=	(⟨*Declaration*⟩ ;)*
⟨*Declaration*⟩	::=	⟨*Typedecl*⟩ | ⟨*Symboldecl*⟩
⟨*TypeArg*⟩	::=	⟨*Name*⟩
⟨*TypeArgs*⟩	::=	⟨*TypeArg*⟩ (, ⟨*TypeArg*⟩)*
⟨*Typedecl*⟩	::=	**type** ⟨*Name*⟩ [[⟨*TypeArgs*⟩]] [= ⟨*Type*⟩]
⟨*Symboldecl*⟩	::=	**declare** ⟨*Identifier*⟩ : [[⟨*TypeArgs*⟩]] ⟨*Type*⟩ [⟨*Assocspec*⟩ ⟨*Precspec*⟩]
⟨*Assocspec*⟩	::=	**prefix** | **infix** | **suffix**
⟨*Precspec*⟩	::=	⟨*Digit*⟩

Some remarks can be already stated at this point about type and symbol declarations in order to clarify the declaration constructions. A type declaration may state more than a type abbreviation, in which the reduced form is specified after the equals symbol, it can also be used to declared a new type name, so no reduced form will appear.

For a symbol declaration the associated type is always present, being optional the specification of associative and precedence information in order to establish the syntactic power of an operator. In both cases it is possible to declare a set of distinct type parameters that specify that the declared name is type parametric, which for symbols allows the introduction of polymorphism in the POLY language.

The scope of the declared names is the entire compilation unit, starting at the point where the name was declared.

For the sake of illustration a list of declarations is shown in the next subsection.

3.2.2 Types

POLY is a strongly typed language. Therefore the declaration of new identifiers involves *type expressions*.

⟨*PrimitiveType*⟩	::=	⟨*TypeL*⟩
⟨*BaseType*⟩	::=	(⟨*Name*⟩ | ⟨*PrimitiveType*⟩) [[⟨*Types*⟩]]
⟨*Type*⟩	::=	⟨*BaseType*⟩ | ⟨*Constructortype*⟩
⟨*Constructortype*⟩	::=	(⟨*Types*⟩) ⟨*BaseType*⟩
⟨*Types*⟩	::=	⟨*Type*⟩(,⟨*Type*⟩)*

As expected type forms consider type literals and identifiers as their basis. From there the construction of new types is possible through the parametrization of a type name or literal, which causes the parameters of the type to be replaced by the types given as arguments. It is also possible to define a compound type which represents the `POLY` constructor form, given by a collection of types for the arguments and a return type, where a compound type may not surface.

The following examples will help to illustrate the mentioned concepts, for both types and declarations.

Examples

⟨*Typedecl*⟩:
type tree;
type product[X,Y];
type prop = **msg**;
type channel[X] = (X)**msg**;
type stack[A] = **list**[A].

⟨*Symboldecl*⟩:
declare qsort:(**list** [**integer**], **list** [**integer**])**msg**;
declare reverse:[X](**list** [X], **list** [X])**msg**;
declare :::[X](X, **list** [X])**list** [X] **infix** 6;
declare pipe:channel[**integer**];
declare pair:[X,Y](X,Y)product[X,Y].

3.2.3 Processes

Process expressions contemplate the parallel composition of processes, the definition, the command, the message form, the restriction construction and the inaction agent.

$$
\begin{array}{lll}
\langle Process \rangle & ::= & \langle Simple \rangle \mid \langle Composition \rangle \\
\langle Simple \rangle & ::= & \langle Inaction \rangle \\
 & \mid & \langle Restriction \rangle \\
 & \mid & \langle Command \rangle \\
 & \mid & \langle Definition \rangle \\
 & \mid & \langle Message \rangle \\
 & \mid & (\langle Process \rangle) \\
\langle Inaction \rangle & ::= & \textbf{inaction} \\
\langle Message \rangle & ::= & \langle Term \rangle \\
\langle Restriction \rangle & ::= & \textbf{new}\ \langle Decls \rangle\ \textbf{in}\ \langle Simple \rangle \\
\langle Command \rangle & ::= & \textbf{com}\ [\ <\ \langle TypeArgs \rangle\ >\ ]\ \langle BasicCommand \rangle \\
\langle Definition \rangle & ::= & \textbf{def}\ [\ <\ \langle TypeArgs \rangle\ >\ ]\ \langle BasicCommand \rangle \\
\langle BasicCommand \rangle & ::= & [\langle Decls \rangle\ \textbf{in}]\ [\langle Input \rangle]\ \langle Test \rangle\ \langle Simple \rangle \\
\langle Decls \rangle & ::= & \langle Ids \rangle : \langle Type \rangle\ (;\langle Ids \rangle : \langle Type \rangle)^* \\
\langle Ids \rangle & ::= & \langle Identifier \rangle\ (,\langle Identifier \rangle)^* \\
\langle Test \rangle & ::= & [] \mid [\langle Process \rangle] \\
\langle Input \rangle & ::= & \langle Term \rangle\ (\&\ \langle Input \rangle)^* \\
\langle Composition \rangle & ::= & \langle Simple \rangle\ (\ |\ \langle Simple \rangle)^*
\end{array}
$$

While the semantics of these language constructions will be explained later on, it is important to mention the scope of the present declarations. For the restriction agent the new identifiers can be present throughout the restrictions continuation. For the definition and command both type parameters identifiers (given by the optional $\langle TypeArgs \rangle$) and new identifiers can be found in the agent's pattern, in the test agent and also in the continuation agent.

3.2.4 Terms

`POLY` terms are no more than the constructor form and it's related specifications, when it comes to operator usage.

$$
\begin{array}{lll}
\langle Term \rangle & ::= & \langle SimpleTerm \rangle \\
 & \mid & \langle Term \rangle\ \langle Binop \rangle\ \langle SimpleTerm \rangle \\
 & \mid & \langle Preop \rangle\ \langle Term \rangle \\
 & \mid & \langle Term \rangle\ \langle Postop \rangle \\
\langle SimpleTerm \rangle & ::= & \langle Name \rangle \\
 & \mid & \langle ValueLiteral \rangle \\
 & \mid & (\langle Term \rangle) \\
 & \mid & \langle Identifier \rangle(\langle Terms \rangle) \\
\langle Terms \rangle & ::= & \langle Term \rangle\ (,\langle Term \rangle)^* \\
\langle Binop \rangle & ::= & \langle Operator \rangle \\
\langle Preop \rangle & ::= & \langle Operator \rangle \\
\langle Postop \rangle & ::= & \langle Operator \rangle
\end{array}
$$

```
declare :::[X](X, list[X])list[X] infix 6;
declare reverse:[X](list[X], list[X])msg;

def <Z> l, r: list[Z] in reverse(l, r) []
new rev:(list[Z], list[Z])msg in
(
  rev(l, nil)
  | def x: Z; l, r: list[Z] in rev(x::l, r) [] rev(l, x::r)
  | com rev(nil, r) [] inaction
)
```

Figure 3.1: Sample POLY code.

```
declare +.:(string, string)string infix 6;
declare word:(string)msg;

word("hello"+." world")
```

Figure 3.2: Operator declaration and usage.

3.3 Sample Code

For the sake of illustration, Figure 3.1 presents a sample compilation unit.

3.4 Implementation

The static checker that was developed for the POLY language makes use of two collaborating tools to capture the presented syntax specification. For the lexical analyzer JLex [Lex] was used while CUP [CUP] ensured the parser. Both specifications made for these tools are a straightforward implementation of the presented syntax, but there are some interesting details regarding operator parsing and error information.

In order to handle operator syntactic power for names declared at the compilation unit level, the parser holds the declared operators in a determined table, which allows the lexical analyzer to check if the symbol string in hand is in fact a declared operator. The problem here is that the precedence of the operator can not be defined at that time, but a simple solution was arranged to deal with this problem: there is a fixed number of precedence values (the chosen value is ten, which probably is more than enough) and the respective token values, leaving to the lexical analyzer the simple task of returning an appropriate operator token, while the parser makes no fuss about it. Figure 3.2 shows the declaration and usage of a string concatenation constructor.

To ensure some helpful information at grammar error level, the implemented static checker reproduces the parser's transition machine, while han-

```
***
Syntax error: Found '|' in line 2, column 0
* Expecting *
infix operator
prefix operator
suffix operator
NAME
integer constant
real constant
String constant
nil
true
false
(
com
def
new
inaction
declare
type
* Context *
declare word:(string)msg;
|
word("hello")
***
```

Figure 3.3: Syntactic error information.

dling the error in order to find out what kind of token was expected by the parser, helping the user to correct the problem. This was one of the efforts made in an attempt to make the static checker as user friendly as possible. Figure 3.3 shows how the information is presented to the user in the form of expected tokens.

In appendix B you can find the complete specification of the grammar for the CUP parser generator, provided as a text output of the grammar that the CUP tool offers.

To close this section, it is important to mention that the grammar builds an abstract syntactic tree while it is parsing the source file, which will be used for the type checking algorithm. A complete listing of the nodes that compose this abstract syntactic tree is shown in the class listing, presented in appendix C, where all the nodes have their class names prefixed with AST - from abstract syntactic tree.

For more information on compiler implementation, regarding these and other related issues, refer to [App98].

Chapter 4

Types

There is a large number of formal methods to ensure that a system behaves correctly with respect to some specification, being the most common the type systems because of their simplicity/effectivity duality in their task of ensuring that determined runtime errors do not occur. For detailed information on type systems consult [Pie02].

Regarding that the communication of processes is a corner stone of the POLY language, it was a central aspect in the development of the type system in order to ensure the absence of a great deal of communication errors, which are the most probable runtime errors in POLY. In this chapter POLY's types are thoroughly explained, firstly through the motivation that stirred their creation and secondly through a formal definition of the type system. To close this chapter a reference is made to implementation issues that seem more relevant to mention.

4.1 Motivation

The better way to introduce the earlier motivation in the development of POLY's type system is to provide an example, like the one present in Figure 4.1, adapted from Figure 2.3, where types would certainly give a helping hand. In this example a complete mess is made in the usage of the concatenation constructor, appearing as a message with one integer argument (present in the test agent) and then in a definition pattern, with the expected three arguments.

```
def left,right,res in concatenation(l,r,res) [] ...
|
def [concatenation(1)] ...
```

Figure 4.1: Malformed program.

```
declare concatenation:
  (list[integer], list[integer], list[integer])msg;

def l,r,res:list[integer] in concatenation(l,r,res) [] ...
|
def [concatenation(1)] ...
```

Figure 4.2: Malformed program with types.

```
declare concatenation:
  (list[integer], list[integer], list[integer])msg;
declare cons:(integer, list[integer])list[integer];

def l,r,res:list[integer] in concatenation(l,r,res) [] ...
|
def ints:list[integer]
  [concatenation(cons(1, nil), cons(2, nil), ints)] ...
```

Figure 4.3: Typing correct program.

The result of such a malformed program is that no successful computation can occur because the agent's pattern is unsatisfiable.

If types were introduced in the language it would be possible for a static checker to reject this kind of situation, being their usage explicit in the POLY language due to two considerations: user simplicity and ambiguity resolution.

The program is again shown in Figure 4.2, now containing type declarations. The problem, easily identified, rests in the message, which form differs completely in respect to the constructor declaration. Presented in Figure 4.3 is a typing correct program.

From this first example one can already identify the first types considered for the POLY language: the expected basic types - for instance the **integer** present in the example - and the compound type forms (for example the (**list**[**integer**], **list**[**integer**], **list**[**integer**])**msg** used as the type of concatenation), being the first essential to any useful language while the second necessary to establish a framework for the language's data constructors.

The other type form present in the poly language is the type constructor, for instance **list**. This type form allows the constructions of types through their parametrization, like in **list**[**integer**], and therefore the suited name for these types is type constructor.

Through the top level declaration construct **type** it is possible to introduce new type constructors in POLY as is shown in Figure 4.4. The declaration present in the example introduces consType as a type constructor, used to create types for list constructors such as the consInt or the consBool.

The existence of type constructors justifies their classification, like the

```
type consType[X]=(X, list [X]) list [X];
declare consInt:consType[integer];
declare consBool:consType[boolean];
```

Figure 4.4: Type constructor.

one that types provide to values, meaning that types will have their own types, referred to as kinds.

Regarding the type constructors, a straightforward classification comes to mind, which is the number of parameters involved, meaning that kinds will be no more than integers that corresponds to the number of parameters of the type constructor. When the arguments replace the parameters the type construction is established, so it's kind is 0. For example, while consType as itself has kind 1, consType[**integer**] has kind 0, because one argument can be applied to the first case while none can in the latter.

Along with parametric types `POLY` supports type parametric values, being an example shown in (4.1) where the declared value represents a list constructor able to handle any kind of lists. This polymorphic operator is very useful, as are all type generic data constructors, being essential to the intended support of type generic algorithms.

$$\textbf{declare}\ \text{cons:}[\text{X}](\text{X},\ \textbf{list}\,[\text{X}])\,\textbf{list}\,[\text{X}]; \tag{4.1}$$

The usage of these constructors is the only exception to the explicit typing present in `POLY`, because they are used without reference to the type argument that is present, therefore requiring it's inference for type checking purposes. An example of this non explicit use of polymorphic values is shown in (4.2), where one can easily notice that the type argument being implicitly provided is **integer**.

$$\text{cons}\,(1,\ \textbf{nil}) \tag{4.2}$$

An implicit notion is present in the usage of these type parametric values, concerning the fact that the arguments must capture all the type parameters to ensure that a valid instantiation can be obtained regarding the used arguments. Therefore, both in type and value top level declaration, this incorrect usage of type parameters must not appear.

An interesting remark to be made about the example shown in (4.2) is the usage of **nil**, which is a polymorphic value, so no explicit specification of the type argument is made. In **nil**'s case the inference of the type argument can be made through the information provided by the placement of **nil**. For instance, in the presented example, the type inferred for the empty list would be **list** [**integer**] because of the implicit association made through the cons type argument. If a substitution is made to this type argument, as is

```
declare concatenation:[X](list[X],list[X],list[X])msg;

def <X> l,r,res:list[X] in concatenation(l,r,res) [] ...
```

Figure 4.5: Type variables.

shown in (4.3), the type determined for the second argument is **list**[**integer**] and therefore **nil**'s type must be the same.

$$([X](X,\ \textbf{list}[X])\textbf{list}[X])[\textbf{integer}] \rightarrow (\textbf{integer},\ \textbf{list}[\textbf{integer}])\textbf{list}[\textbf{integer}] \quad (4.3)$$

An example that illustrate the correct form, regarding the referred type coherence, of constructor cons is shown in (4.4), while an example of incorrect cons usage is given in (4.5). The idea present is that nested constructors must have return types equivalent to the argument types where they are placed, as could be expected.

$$\text{cons}(1,\ \text{cons}(2,\ \textbf{nil})) \quad (4.4)$$

$$\text{cons}(1,\ \text{cons}(\textbf{true},\ \textbf{nil})) \quad (4.5)$$

An intuition on this type coherence present in the constructors can be provided through the presented examples, and it simply refers to the chosen declaration of the constructor. In the cons case the intent is to create a list constructor that builds lists of elements of any kind, but between the elements the type remains the same.

A remark to be made at this point is the fact that a message must be type grounded, meaning that all type parameters must be instantiated, considering the referred type coherence. An example that shows an incorrect message term is shown in (4.6), where either the concatenation type parameter or the **nil**'s type parameters can not be instantiated.

$$\text{concatenation}(\textbf{nil},\ \textbf{nil},\ \textbf{nil}) \quad (4.6)$$

To fulfill the support of type generic algorithms, along with polymorphic constructors, comes the need of type variables present in the definition and command, used to confer polymorphism to acceptance patterns. An example of the usage of these type variables is shown in Figure 4.5.

Type variables work somewhat like value variables: they are instantiated during the pattern matching procedure and the types that instantiate them will work as any other in the continuation agent.

In the example shown, the idea is to provide a type generic list concatenation algorithm, which is an instance of an algorithm that makes sense to be type independent, in this case, in regard to the type of the list's elements.

```
declare concatenation:[X](list[X], list[X], list[X])msg;
declare cons:[X](X, list[X])list[X];

def l1, l2, l3:list[integer] in concatenation(l1, l2, l3) [] ...
|
def ints: list[integer] in [concatenation(cons(1, nil), cons(2,
    nil), ints)] ...
|
def bools: list[boolean] in [concatenation(cons(true, nil), cons
    (false, nil), bools)] ...
```

Figure 4.6: Dynamic versus static type checking.

To obtain the intended idea, it is required that the concatenation constructor declaration makes it type parametric, as it is in the illustrated declaration, which ensures that the constructor is able to handle any kind of lists. Secondly it is necessary to consider the type variable present in the definition, which will stand for an universal type, meaning that the pattern term is matchable to any concatenation message regardless of the type of the elements of the lists.

There is a set of problems that surface when developing the support for this language feature, namely an important trade-off between dynamic and static type checking, also referable as runtime and compile time type checking. In the first case it is possible to maintain a complete freedom in the usage of this feature while in the second there are some restrictions that have to be made to ensure the correct behavior of the system. The next section covers the details that concern this problem.

4.1.1 Pattern matching

Figure 4.6 provides an example of a program that shows where the problem lies, which is in the declaration of the variables used in the acceptance pattern. The declarations present in the first definition, illustrated in the example, state that the pattern is only receptive to integer lists, but looking at the usage examples, one can notice that there is no problem in the boolean usage, other than it can not be handled in the referred definition.

If dynamic type checking is used, the handling of the boolean lists would simply be rejected by the presented definition, due to runtime verification of the types involved in the pattern matching, but this procedure is already a heavy task and overloading it would only make it more inefficient. On the other hand, static checking can't reject any of the presented processes but there will be trouble when the booleans are handled, because there is no type information at runtime, meaning that restrictions will have to be placed in order to establish a system free from runtime type verification.

The basic idea is that if the acceptance pattern type checks correctly

```
declare concatenation:[X](list[X], list[X], list[X])msg;
declare cons:[X](X, list[X])list[X];

def <X> l1, l2, l3:list[X] in concatenation(l1, l2, l3) [] ...
|
def ints: list[integer] in [concatenation(cons(1, nil), cons(2,
    nil), ints)] ...
|
def bools: list[boolean] in [concatenation(cons(true, nil), cons
    (false, nil), bools)] ...
```

Figure 4.7: Program suitable for static checking.

than all messages that also type check correctly are matching candidates to the pattern, meaning that no type restrictions are present, thus ensuring the absence of runtime errors due to incompatible types present in the matching procedure. The previous example is again presented in Figure 4.7, reconfigured in order to be validated through the static type checker.

Using the type variable at the definition level, it is ensured that the desired property stands in the presented example, because the type variable provides the desired universal polymorphism in the acceptance pattern, being the most reasonable usage when the intent is to specify a type generic algorithm.

However, it is still possible to define particular cases through the usage of concrete values or upper level variables in the pattern, because values and instantiated variables can ground the pattern term to a determined type. The reason behind this argument is that no message having the same values or instantiated variables, and therefore no message that is a matching candidate, will have a different type. An example of value usage in the acceptance pattern is shown in Figure 4.8, where the concatenation construct appears restricted to list of integer handling due to it's first argument. Looking at the example, one can easily realize that no other concatenation message other than one holding integer lists is a matching candidate. In such cases the variables must be declared accordingly, meaning that no type variables must be present, only the particular types, as in the example shown.

This property, however, does not stand for polymorphic values, because different messages containing the same value might have different types involved. If, for instance, **nil** is used in a pattern it will not be able, by itself, to ground the pattern term when it comes to the elements of the list involved in that argument, due to the fact that it is possible that this polymorphic value surfaces in different messages having different types.

Like polymorphic values, variables do not support this property, being the justification the same: distinct messages can hold different types and be a matching candidate with respect to a variable, if the declared type of the

```
declare concatenation:[X](list[X], list[X], list[X])msg;
declare cons:[X](X, list[X])list[X];

def l2, l3:list[integer] in concatenation(cons(1,nil), l2, l3)
     [] ...
|
def ints: list[integer] in [concatenation(cons(1, nil), cons(2,
    nil), ints)] ...
|
def bools: list[boolean] in [concatenation(cons(true, nil), cons
    (false, nil), bools)] ...
```

Figure 4.8: Usage of values in the acceptance pattern.

```
declare op:[X,Y](X,Y)msg;

def <Z> a,b:Z in op(a,b) [] ...
| op(1,true)
```

Figure 4.9: Type variable incorrect usage in a single term.

variable is disregarded.

Hence variable usage in the acceptance pattern will be an important focus of the pattern matching type verification. Their declaration must be coherent in regard to their usage, meaning that, to assure the non establishment of type restrictions, the variables must be declared in such a way that any value, contained in a matching candidate message, is supported by the type, therefore justifying the usage of type variables.

Regarding the correct usage of type variables, in such a way that they stand for the generic type intended, an important verification must be made to guarantee the independence between them. The example shown in Figure 4.9 illustrates one case where there is an dependency established between the type variables. The message present in the example shows why variables a and b must not share the type variable Z, because the op constructor supports different types on it's arguments, so a type restriction is the one thing making the matching impossible.

Another example can be found in Figure 4.10 where the type information is being shared in two different pattern terms. Once again type-blind matching is possible, but the type variable would have to take two different values, which is absurd.

While the properties referred above report to the minimal level of generality that the type variables must support, another set of restrictions was established on the usage of type variables, regarding their semantic interpretation, that enforce the usage of type variables with the precise level of generality required, instead of allowing a free usage of their generic power.

```
declare op:[X](X)msg;

def <Z> a,b:Z in op(a) && op(b) [] ...
| op(1)
| op(true)
```

Figure 4.10: Type variable incorrect usage in different terms.

```
declare op:[X](list[X])msg;

def <Z> a:Z in op(a) [] ...
| def <Z> b:list[Z] in op(b) [] ...
| op(cons(1,nil))
| op(cons(true,nil))
```

Figure 4.11: Type variable generality.

An example is shown in Figure 4.11 where the first definition is typing incorrect due to a declaration of a variable of a unnecessary level of generality, in contrast with the second definition that declares a variable that is sufficiently generic.

Having presented the intuition required for the comprehension of `POLY`'s types, the next sections complete their description as well as formally present the the type system.

4.1.2 `POLY`'s types

To strengthen the ideas presented previously this section illustrates the allowed type forms in `POLY`, which are roughly presented in the grammar presented in (4.7).

$$\begin{aligned} BT &::= \text{b} \in \texttt{Basic} \mid \text{id} \\ AT &::= BT[CT_1, \ldots, CT_n] \ (n \geq 0) \\ CT &::= (CT_1, \ldots, CT_n)AT \mid AT \mid X \\ TS &::= [X_1, \ldots, X_n]CT \ (n \geq 0) \end{aligned} \tag{4.7}$$

Starting by the last production TS, it shows that a type scheme may (or not) be a type constructor, in which case a set of type parameter names $[X_1, \ldots, X_n]$ is priorly given. Production CT refers to compound types that consisted in a sequence of compound types $(CT_1, \ldots, CT_n)$, followed by a return type that must be a basic type, possibly with type construction parameters, as is shown in production AT. Production CT production also resolves into a type parameter (X) or simply an atomic type (AT). Finally, a basic type (BT) is either a predefined type (`Basic`), like **integer** or **boolean**, or an identifier represented by id.

The following examples may help to clarify the admissible type constructions.

- **integer**

  A basic type.

- **list** [**boolean**]

  A type constructor, representing list of booleans.

- [X, Y](X, (Y)**msg**)**msg**

  A parametric compound type constructor, being `X` and `Y` the type parameters.

- [X](X, **list** [X]) **list** [X]

  The list constructor type.

4.2 Rule specification

Before presenting the type system itself, keeping in mind that what is intended is a formal type system, it is necessary to fully illustrate the notation of the type rules, which are the basis of the type system.

The basic element in a type rule is a judgment, represented as shown below, where $\Im$ is an assertion, which form varies as is discussed further, and Γ is the environment of the evaluation, where are present associations between names and their types or their kinds and reductions, depending if they are a value or a type, respectively.

$$\Gamma \vdash \Im$$

Using the presented notation for the judgment, it is possible to show the general form of a type rule.

$$\begin{array}{l}(\textit{Rule name}) \\ \dfrac{\Gamma_1 \vdash \Im_1 \cdots \Gamma_n \vdash \Im_n}{\Gamma \vdash \Im} \\ (\text{Annotations})\end{array}$$

Above the line appear the premises of the rule ($\Gamma_i \vdash \Im_i$) while the conclusion appears below the line. The idea is that the conclusion is valid if all the premises also are. This general form is present in the following type system, but there are some details concerning the different forms of assertions that need to be cleared at this stage, so a more exhaustive list of judgment form is shown.

4.2.1 Judgment notation

- **Declaration judgments**

$$\Gamma \vdash D \Rightarrow \Gamma'$$

  This form of judgment is used for the top level name declaration in the `POLY` language (D), specified by **declare** or **type**, where the assertion produces a new environment, represented by Γ'.

- **Agent judgments**

$$\Gamma \vdash A \quad OK$$

  This notation is used in agent verification rules and it simply represents that a determined agent, represented by A, is well formed.

- **Kind judgments**

$$\Gamma \vdash T :: K \Rightarrow T'$$

  This notation is used when the evaluation is being made on a type, represented by T, so the necessary information to be retrieved by the assertion is the kind of the type and a reduction, represented by K and T', respectively.

- **Term judgments**

$$\Gamma \vdash u : T$$

  Here the idea is to denote that a term, represented by u, has a determined type, given by T.

- **Pattern term judgments**

$$\Gamma \vdash term(u) \Rightarrow (S, T, Z)$$

  The presented judgement is used to establish not only the type of term u, represented by T, but also to conceive a type identification equation system, identified by S, regarding u's inner structure, and also the variable set, represented by Z, that was obtained while conceiving S.

4.3 Type system

Finally, this section presents the formal type system through a set of rules that use the notations explained in the previous section.

4.3.1 Unit verification

As explained earlier a `POLY` program is composed of a sequence (possibly empty) of declarations and possibly an agent. The rules that specify how the unit verification is handled are presented in (4.8).

$$\begin{array}{l}(\textit{Unit}) \\ \dfrac{\Gamma \vdash D \Rightarrow \Gamma' \quad \Gamma' \vdash A \Rightarrow OK}{\Gamma \vdash D;\ A \Rightarrow OK} \\ \\ (\textit{Declaration list}) \\ \dfrac{\Gamma \vdash D_1 \Rightarrow \Gamma' \quad \Gamma' \vdash D_2 \Rightarrow \Gamma'' \cdots \Gamma^{n-1} \vdash D_n \Rightarrow \Gamma^n}{\Gamma \vdash D_1; \cdots ; D_n \Rightarrow \Gamma^n}\end{array} \tag{4.8}$$

The rule, identified by *Unit*, simply illustrates that a `POLY` compilation unit is a valid program when the declarations are correctly typed and the agent is well formed regarding them. The second rule, *Declaration list*, simply constructs the new environment through the evaluation of all the declarations present in the program, extending the environment declaration by declaration.

4.3.2 Declaration verification

The set of rules illustrated in (4.9) refer to the top level declaration of types and operators in `POLY`, so the judgment form that will appear will be the Declaration judgment.

$$\begin{array}{l}(\textit{Type declaration}) \\ \dfrac{}{\Gamma \vdash \textbf{type}\ \mathrm{id}[X_1, \ldots, X_n] \Rightarrow \Gamma, \mathrm{id} :: n} \\ (\forall i, j \quad i \neq j \Rightarrow X_i \neq X_j) \\ \\ (\textit{Type declaration with abbreviation}) \\ \dfrac{\Gamma, X_1 :: 0, \ldots, X_n :: 0 \vdash T :: 0 \Rightarrow T'}{\Gamma \vdash \textbf{type}\ \mathrm{id}[X_1, \ldots, X_n] = T \Rightarrow \Gamma, \mathrm{id} :: n = [X_1, \ldots, X_n]T'} \\ (X_1, \ldots, X_n \text{ are present in } T \text{ in argument positions.}) \\ \\ (\textit{Symbol declaration}) \\ \dfrac{\Gamma, X_1 :: 0, \ldots, X_n :: 0 \vdash T :: 0 \Rightarrow T'}{\Gamma \vdash \textbf{declare}\ \mathrm{id} : [X_1, \ldots, X_n]T \Rightarrow \Gamma, \mathrm{id} : [X_1, \ldots, X_n]T'} \\ (X_1, \ldots, X_n \text{ are present in } T \text{ in argument positions.})\end{array} \tag{4.9}$$

The first rule, identified by *Type declaration*, simply shows that when a new name is declared as a type, with a given set (possibly empty) of parameters, a new environment is created by adding the association between the declared type and it's kind to the previously established environment.

The *Type declaration with abbreviation* rule handles the declaration of a type name given as an abbreviation of a type scheme. The type scheme, represented by T, is checked both for its possible reduction and, evidently, for its validity in an extended environment, containing the declaration of the type parameters. Having the premise valid the association between the declared name and both the kind and the reduced type scheme is established as an extension of the old environment. The annotation presented refers to the fact that all type parameters must be present in argument positions in T, as could be expected, because it makes no sense to declare a data constructor with a return type that is not present in it's arguments. An example of a verification using this rule is given in (4.10).

$$\frac{\dfrac{\dots}{\{\emptyset, X :: 0\} \vdash (\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}] :: 0 \Rightarrow (\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]}}{\begin{array}{c}\emptyset \vdash \mathbf{type}\ \mathrm{consType}[\mathrm{X}] = (\mathrm{X}, \mathbf{list}[\mathrm{X}])\mathbf{list}[\mathrm{X}] \Rightarrow \\ \{\emptyset, \mathrm{consType} :: 1 = [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\}\end{array}} \tag{4.10}$$

Finally the *Symbol declaration* rules shows the same form present in its predecessor, only differing because it is an operator declaration, not a type, so there is no kind present in the association. Illustration (4.11) shows a verification example.

$$\frac{\dfrac{\dots}{\begin{array}{c}\{\mathrm{consType} :: 1 = [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\} \vdash \mathrm{consType}[\mathbf{integer}] :: 0 \Rightarrow \\ (\mathbf{integer}, \mathbf{list}\,[\mathbf{integer}])\,\mathbf{list}\,[\mathbf{integer}]\end{array}}}{\begin{array}{c}\{\mathrm{consType} :: 1 = [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\} \vdash \mathbf{declare}\ \mathrm{consInt:consType}[\mathbf{integer}] \Rightarrow \\ \{\cdots, \mathrm{consInt} : (\mathbf{integer}, \mathbf{list}\,[\mathbf{integer}])\,\mathbf{list}\,[\mathbf{integer}]\}\end{array}} \tag{4.11}$$

4.3.3 Agent verification

In this section the rules, shown in (4.12), refer to the type checking of an agent so the judgment form used is the agent judgment that simply states, as mentioned earlier, that a determined agent construction is well formed.

$$\frac{}{\Gamma \vdash \textbf{inaction} \quad OK} \quad (\textit{Inaction})$$

$$\frac{\Gamma \vdash P \quad OK \quad \Gamma \vdash Q \quad OK}{\Gamma \vdash P \mid Q \quad OK} \quad (\textit{Parallel composition})$$

$$\frac{\Gamma, v_1 : T_1, \ldots, v_n : T_n \vdash P \quad OK}{\Gamma \vdash \textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P \quad OK} \quad (\textit{Restriction})$$

$$\frac{\Gamma \vdash M : \textbf{msg}}{\Gamma \vdash M \quad OK} \quad (\textit{Message})$$
(M is type grounded)

($\textit{Definition}$)

$$\begin{array}{c}
replace(\{M_1, \ldots, M_t\}, \{v_1, \ldots, v_k\}) \Rightarrow \\
(\{M'_1, \ldots, M'_t\}, \{v'_1, \ldots, v'_m\}, \gamma) \\
\Gamma, Y_1 :: 0, \ldots, Y_m :: 0, v'_1 : Y_1, \ldots, v'_m : Y_m \vdash term(M'_1) \Rightarrow \\
(S_1, \textbf{msg}, Z_1) \\
\vdots \\
\Gamma, Y_1 :: 0, \ldots, Y_m :: 0, v'_1 : Y_1, \ldots, v'_m : Y_m \vdash term(M'_t) \Rightarrow \\
(S_t, \textbf{msg}, Z_t) \\
\sigma = mgu_n(S_1 \cup \ldots \cup S_t, \{\overline{Y}\} \cup Z_1 \cup \ldots \cup Z_t) \\
\varphi = mgu_n(\bigcup_{q:Y_q \in dom(\sigma) \wedge \exists_l . \gamma(v'_q) = v_l} \{T_l = \sigma(Y_q)\}, \{\overline{X}\}) \\
\Gamma, X_1 :: 0, \ldots, X_n :: 0, v_1 : T_1, \ldots, v_k : T_k \vdash P \quad OK \\
\Gamma, X_1 :: 0, \ldots, X_n :: 0, v_1 : T_1, \ldots, v_k : T_k \vdash Q \quad OK \\
\hline
\Gamma \vdash \textbf{def} < X_1, \ldots, X_n > v_1 : T_1; \ldots; v_k : T_k \textbf{ in } M_1 \& \ldots \& M_t[P]Q \quad OK
\end{array}$$

($v_1, \ldots, v_k$ do not occur as heads of $M_1, \ldots, M_t$)
($\{Y_1, \ldots, Y_m\}$ is a set of fresh names)
($\forall_{i \in 1,\ldots,n} \exists_j \ \varphi(X_i) = W_j \quad \forall_{i,j \in 1,\ldots,n} \ \varphi(X_i) = \varphi(X_j) \Rightarrow i = j$)

(4.12)

The *Inaction* rule is trivial: the agent that does nothing is well formed, as could be expected. Referring to the parallel composition of agents, the second rule, identified by *Parallel composition*, illustrates that it is well formed if the agents that compose it are well formed. The third rule, *Restriction*, shows that a restriction is well formed when its continuation also is, evaluated in an extended environment containing the new variable declarations.

Following comes the *Message* rule that simply illustrates that a message must resolve to predefined type **msg** - through term verification - to ensure that the agent composed by it is well formed. There is, however, an im-

portant remark to be made regarding the annotation shown that is the fact that a message must not have, in all it's imbricated terms and in itself, type parameters that have not been instantiated.

Finally comes the *Definition* rule, being the command such an identical case that the rule would only differ in the specified keyword, so only the rule shown is the one for the definition.

This last rule states that a definition agent is well formed if the pattern, the test agent and the continuation agent are typing coherent. In order to state the typing correctness of the pattern a possible verification goes by inferring the types of the variables, looking only at the pattern to do so, and then crosschecking them with the declarations present in the definition.

If the types inferred for the variables are equivalent to the ones declared, it is ensured that no typing restrictions are made to the pattern matching operation, because the inference can only bring up the most generic types possible, so if the declarations are equivalent then they do not enforce any typing restrictions.

The inference of the types of the variables is made effective by evaluating all the pattern terms considering a replacement of each variable occurrence with a fresh variable associated with a type variable (refer to section 4.5), being the latter used to hold the inferred information. The first premise of the rule specifies this substitution of every occurrence of a fresh variable with a fresh name while the following t rules handle the construction of equation systems, one per each pattern term, where the equations represent the identification of declared types versus used types (the details are presented in section 4.3.6). These rules also state that the return type encountered for the pattern term is the expected **msg** predefined type.

The following premise states that there must exist a most general unifier (refer to section 4.4) for the equation system resulting from the union of the ones mentioned above, considering as variables the type variables associated with the introduced fresh names and the type parameters of encountered polymorphic constructs. As expected, the solution of the most general unification algorithm is a substitution for all the considered variables such that the system is solved, which means that each type variable has an expected type that can be found in the value that the substitution enforces on it, providing the intended inferred type information.

The image set of this substitution is composed of two sorts of elements, namely any kind of valid type construction or special type variables that represent that through the context information it was not possible to determine a ground type, appearing both in isolation or inside a type construction. Implicit in this reasoning is the fact that nested constructors that have type dependencies will have this relation present wether through the assignment of their type parameters to the same type construct, grounded or not, or simply through an identification equation if no type parameters are involved.

With the information retrieved in the previous step it is possible to check

```
declare op:[X](X,X)msg;
declare other_op:[X,Y](X,Y)msg;

def <X, Y> x:X;y:Y in op(x,y) [] ...
|
def <X> x,y:X in other_op(x,y) [] ...
```

Figure 4.12: Incorrect usage of acceptance patterns.

if all the occurrences of the variable are coherent with themselves but also with the declared type through the the determining of a most general unifier to an equation system where are present the identifications between each inferred information for each fresh type variable and the type information present at the declaration level. The variables considered for the solution of the system are the type variables present at the definition level, being a valid solution a substitution that for each type variable associates the referred special type variables that represent the absence of a restriction to a ground type, or in other words, it basically states the intended idea that type variables must not be restricted to any sort of ground type.

An important remark to be made about the agent's pattern is related to the first annotation made in the type rule, which concerns the fact that the variables declared in the definition must not surface as heads of any constructor terms present in the agent's pattern, which would make it impossible to satisfy, because there can be no constructions having that identifier as their head in any message, due to it's declaration being present only at the time of the introduction of the definition.

It is also essential to ensure that all type variables belong to the domain of the substitution, meaning that the pattern matching captures all type information, and that they all have distinct values in the image of the substitution, representing the absence of associations between type variables, as specified in the third annotation.

The reason that the referred correspondence is strictly one to one is a condition necessary to the assurance that no type restrictions are made: If two definition type variables were to have a correspondence to the same type special constant, a restriction would be established because the two type variables would not be independent, thus, not as generic as required; if two special constants were to have an association with a single definition type variable, that would mean that the type variable could assume two different types at the same time, which, of course, is impossible.

Presented in Figure 4.12 are two examples of why the referred condition is imposed. In the first definition type variables X and Y are used to capture a single possible type value. In the second definition, the opposite occurs: a constructor used to capture two values of any type is being used in the pattern with two variables of the same type, which is no more than a par-

ticular case of the specified constructor and therefore a matching restriction is established.

An interesting remark is that the non establishment of restrictions between pattern terms is ensured by the referred condition of strict correspondence, because from different pattern terms must surface different special type variables, thus providing the guarantee that no type variables are shared between pattern terms.

Having guaranteed the correctness of the agent's pattern, it is also necessary to state that the test agent and the continuation agent are well formed, being this verification made in an environment extended with the new variable declarations and also with the type parameters present in the definition agent, as is shown in the two final premises of the rule.

The usage of generic types gives the intended support for the non establishment of restrictions at the pattern matching level, therefore making dynamic type checking unnecessary. The example presented in Figure in (4.17) can help to ground the referred ideas involving pattern term verification.

$$\frac{\cdots}{\begin{array}{c} replace(\{\text{op(cons(x, lst))}\}, \{\text{x, lst}\}) \Rightarrow \\ (\{\text{op(cons(x', lst'))}\}, \{\text{x', lst'}\}, \{\text{x'} \leftarrow \text{x, lst'} \leftarrow \text{lst}\}) \end{array}} \tag{4.13}$$

$$\frac{\cdots}{\begin{array}{c} \{\text{op} : [V](\,\textbf{list}\,[\text{V}])\textbf{msg}, \text{cons} : [Z](\text{Z},\ \textbf{list}\,[\text{Z}])\,\textbf{list}\,[\text{Z}], \\ Y_1 :: 0, Y_2 :: 0, \text{x'} : Y_1, \text{lst'} : Y_2\} \vdash \\ term(\text{op(cons(x', lst'))}) \Rightarrow \\ (\{Z = Y_1; \textbf{list}\,[Z] = Y_2; \textbf{list}\,[V] = \textbf{list}\,[Z]\}, \textbf{msg}, \{Z, V\}) \end{array}} \tag{4.14}$$

$$\{V \leftarrow W'\} \circ \{W \leftarrow W'\} \circ \{Y_2 \leftarrow \textbf{list}\,[W]\} \circ \{Z \leftarrow W\} \circ \{Y_1 \leftarrow W\} = mgu_n(\{Z = Y_1; \textbf{list}\,[Z] = Y_2; \textbf{list}\,[V] = \textbf{list}\,[Z]\}, \{Z, V\} \cup \{Y_1, Y_2\}) \tag{4.15}$$

$$\{X \leftarrow W'\} = mgu_n(\{X = W'; \textbf{list}\,[X] = \textbf{list}\,[W']\}, \{X\}) \tag{4.16}$$

$$\frac{(\ 4.13)\quad (\ 4.14)\quad (\ 4.15)\quad (\ 4.16)\quad \cdots}{\begin{array}{c} \{\text{op} : [V](\,\textbf{list}\,[\text{V}])\textbf{msg}, \text{cons} : [Z](\text{Z},\ \textbf{list}\,[\text{Z}])\,\textbf{list}\,[\text{Z}]\} \vdash \\ \textbf{def} < X > \text{x:X; lst} : \textbf{list}\,[\text{X}]\ \textbf{in}\ \text{op(cons(x, lst))}\ [] \cdots \quad OK \end{array}} \tag{4.17}$$

4.3.4 Kind verification

The rules for kind verification, presented in (4.18), are used when it is necessary to validate a type construction, so there will be kind judgments present, as they were shown above.

$$\frac{}{\Gamma, T :: K \vdash T :: K \Rightarrow T} \; (Id)$$

$$\frac{}{\Gamma, T :: K = T' \vdash T :: K \Rightarrow T'} \; (Abbreviation)$$

$$\frac{\mathrm{b} :: K \in \mathrm{TK}}{\Gamma \vdash \mathrm{b} :: K \Rightarrow \mathrm{b}} \; (Predefined)$$

$$\frac{\Gamma \vdash T_1 :: 0 \Rightarrow T_1' \cdots \Gamma \vdash T_n :: 0 \Rightarrow T_n' \quad \Gamma \vdash T :: 0 \Rightarrow T'}{\Gamma \vdash (T_1, \ldots, T_n)T :: 0 \Rightarrow (T_1', \ldots, T_n')T'} \; (Compound\ type)$$
$$(T' \in AT)$$

$$\frac{\Gamma \vdash U :: n \Rightarrow U \quad \Gamma \vdash T_1 :: 0 \Rightarrow Z_1 \cdots \Gamma \vdash T_n :: 0 \Rightarrow Z_n}{\Gamma \vdash U[T_1, \ldots, T_n] :: 0 \Rightarrow U[Z_1, \ldots, Z_n]} \; (Parametric)$$

$$\frac{\Gamma \vdash T :: n \Rightarrow [X_1, \ldots, X_n]U \quad \Gamma \vdash T_1 :: 0 \Rightarrow Z_1 \cdots \Gamma \vdash T_n :: 0 \Rightarrow Z_n}{\Gamma \vdash T[T_1, \ldots, T_n] :: 0 \Rightarrow U\{X_1 \leftarrow Z_1\} \cdots \{X_n \leftarrow Z_n\}} \; (Parametric\ abbreviation) \tag{4.18}$$

The first rule, identified by *Id*, is used when a type, represented by T, is being evaluated. With respect to the information present in the environment the assertion determines that the type has kind K and that it is completely reduced. This latter condition is not present in the type given in rule *Abbreviation*, where the information provided by the environment indicates that the type has a reduction, identified by T', and kind K. The existence of this reduced form is explained because of the possibility to declare types as abbreviations of any type scheme, which is the case presented in (4.19).

$$\frac{}{\{\mathrm{consType} :: 1 = [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\} \vdash \mathrm{consType} :: 1 \Rightarrow [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]} \tag{4.19}$$

Rule *Predefined* illustrates the verification of a type name that is present in the basic type forms, such as **integer** or **list**. The conclusion is similar to the *Id* rule because both refer to reduced types with a determined kind.

The fourth rule, identified by *Compound type*, shows the verification of a compound type. The premises state that all the types of the arguments of the compound type must be well formed and that all must be a type, meaning that no type constructors are present. If the premises are valid, the

reduction to the compound type is established through the composition of the reduced forms of all the types of the arguments. The judgment concludes this reduced form and also that the compound type has kind 0, because it is also a type, not a type constructor.

To validate a parametric type scheme come the two final rules, which are *Parametric* and *Parametric abbreviation*. The first one concerns a type construction involving a reduced type form, so the necessary premises are that the type must have a kind correspondent to the number of arguments being applied and that the arguments are all well formed, being the type evaluated for the parametric type scheme very similar to it's initial form but considering the reduced forms found for it's arguments, as can be noticed in (4.20).

$$\frac{\dfrac{}{\emptyset \vdash \textbf{list} :: 1 \Rightarrow \textbf{list}}\,(\textit{Predefined}) \quad \dfrac{}{\emptyset \vdash \textbf{integer} :: 0 \Rightarrow \textbf{integer}}\,(\textit{Predefined})}{\emptyset \vdash \textbf{list}\,[\textbf{integer}] :: 0 \Rightarrow \textbf{list}\,[\textbf{integer}]} \tag{4.20}$$

The last rule has a similar form to it's predecessor but differs in one important detail: the type constructor involved is an abbreviation. The premises for the rule are essentially the same, it is in the conclusion that the difference is handled. Because the type has a reduced form it is necessary to apply a substitution of the parameters present in the reduction type scheme to the intended types, determined by the reduction of the applied type arguments, hence obtaining the desired construction. The conclusion states not only the reduced form of the parametric type given, but also, that it corresponds to a type, not a type constructor. An example is shown in (4.21).

$$\frac{\dfrac{}{\{\text{consType} :: 1 = [X](\text{X},\textbf{list}\,[\text{X}])\,\textbf{list}\,[\text{X}]\} \vdash \text{consType} :: 1 \Rightarrow [X](\text{X},\textbf{list}\,[\text{X}])\,\textbf{list}\,[\text{X}]}\,(\textit{Abbreviation}) \quad \dfrac{}{\{\cdots\} \vdash \textbf{integer} :: 0 \Rightarrow \textbf{integer}}\,(\textit{Predefined})}{\begin{array}{c}\{\text{consType} :: 1 = [X](\text{X},\textbf{list}\,[\text{X}])\,\textbf{list}\,[\text{X}]\} \vdash \\ \text{consType}[\textbf{integer}] :: 0 \Rightarrow (\textbf{integer},\textbf{list}\,[\textbf{integer}])\,\textbf{list}\,[\textbf{integer}]\end{array}} \tag{4.21}$$

4.3.5 Term verification

In this section the evaluation is being made on constructions that can be either messages or any other that can be found as arguments in a message form. The notation encountered for the judgment is the term judgment. The rules for this kind of verification are illustrated in (4.22).

(*Value literal*)

$$\frac{}{\Gamma \vdash \mathrm{v} : T}$$

(v is a value literal and T is the associated type)

(*Identifier*)

$$\frac{}{\Gamma, \mathrm{id} : T \vdash \mathrm{id} : T}$$

(*Constructor*)

$$\frac{\Gamma \vdash \mathrm{id} : [X_1, \ldots, X_k](T_1, \ldots, T_n)T \quad \Gamma \vdash \mathrm{u}_1 : \gamma(T_1) \cdots \Gamma \vdash \mathrm{u}_n : \gamma(T_n)}{\Gamma \vdash \mathrm{id}(\mathrm{u}_1, \ldots, \mathrm{u}_n) : \gamma(T)}$$

($X_1, \ldots, X_k$ are fresh names)
($\exists \gamma . dom(\gamma) = \{X_1, \ldots, X_k\}$)

(4.22)

The rule identified by *Value literal* is quite simple and only states that a determined value, belonging to the predefined values of the language, has a given type, like **true** would have type **boolean**. The *Identifier* rule is also very simple and states that a determined identifier has a type determined by the association present in the environment.

The third rule, namely *Constructor*, handles the verification of a compound term, and it states that it's validity is ensured if the name used as the head of the term has a corresponding type respectively to it's usage and that all the used arguments have also corresponding types to the ones present in the associated type, attending to a valid substitution. The term resolves to the type identified as it's return type, once again regarding the valid substitution, which refers to a valid instantiation of the type parameters declared for the compound type. The example shown in (4.23) refers to an evaluation, using this rule.

$$\frac{\dfrac{\dfrac{}{\{\mathrm{cons} : [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\} \vdash \mathrm{cons} : [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]}\,(\textit{Identifier}) \quad \dfrac{}{\{\cdots\} \vdash 1 : \mathbf{integer}}\,(\textit{Value literal}) \quad \dfrac{}{\{\cdots\} \vdash \mathbf{nil} : \mathbf{list}\,[\mathbf{integer}]}\,(\textit{Value literal})}{\{\mathrm{cons} : [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\} \vdash \mathrm{cons}(1, \mathbf{nil}) : \mathbf{list}\,[\mathbf{integer}]}$$

(4.23)

4.3.6 Pattern term verification

The rules present in this section, illustrated in (4.24), refer to constructor verification when they occur in an agents acceptance pattern, so the pattern term judgment is present.

$$\frac{\begin{array}{c}(\textit{Pattern constructor})\\ \Gamma \vdash \mathrm{id} : [Z_1, \ldots, Z_k](T_1, \ldots, T_n)T \\ \Gamma \vdash \mathit{term}(u_1) \Rightarrow (S_1, U_1, V_1) \cdots \Gamma \vdash \mathit{term}(u_n) \Rightarrow (S_n, U_n, V_n)\end{array}}{\begin{array}{c}\Gamma \vdash \mathit{term}(\mathrm{id}(u_1, \ldots, u_n)) \Rightarrow \\ (S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, T, \\ \{Z_1, \ldots, Z_k\} \cup V_1 \cup \cdots \cup V_n)\end{array}}$$
($Z_1, \ldots, Z_n$ are fresh names)

$$\frac{(\textit{Pattern identifier})}{\Gamma, \mathrm{id} : [Z_1, \ldots, Z_n]T \vdash \mathit{term}(\mathrm{id}) \Rightarrow (\emptyset, T, \{\overline{Z}\})} \tag{4.24}$$

$$\frac{(\textit{Pattern value literal})}{\Gamma \vdash \mathit{term}(\mathrm{v}) \Rightarrow (\emptyset, T, \{\overline{Z}\})}$$
(v is a value literal and $[Z_1, \ldots, Z_n]T$ is the associated type)

Rule *Pattern identifier* simply returns the type associated to the identifier and rule *Pattern value literal* returns the corresponding type associated with the given value literal, being also established, for both rules, empty sets representing the non existing type identification equations, and the sets of type parameters that were encountered.

Rule *Pattern constructor* looks at the declared type for the term head identifier and forms the equations corresponding to used versus declared types, returning an accumulated equation system, because inner terms might have formed equation system themselves, and the term's return type, as well as the set of type parameters encountered in the arguments along with the ones present in the constructor being evaluated, as can be seen in (4.25).

$$\frac{\begin{array}{c}\dfrac{(\textit{Identifier})}{\{\mathrm{cons} : [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\}} \\ \vdash \mathrm{cons} : [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\end{array} \quad \begin{array}{c}(\textit{Pattern value literal}) \\ \{\cdots\} \vdash \mathit{term}(1) \Rightarrow \\ (\emptyset, \mathbf{integer}, \emptyset)\end{array} \quad \begin{array}{c}(\textit{Pattern value literal}) \\ \{\cdots\} \vdash \mathit{term}(\mathbf{nil}) \Rightarrow \\ (\emptyset, [Z]\,\mathbf{list}\,[\mathrm{Z}], \{Z\})\end{array}}{\begin{array}{c}\{\mathrm{cons} : [X](\mathrm{X}, \mathbf{list}\,[\mathrm{X}])\,\mathbf{list}\,[\mathrm{X}]\} \vdash \mathit{term}(\mathrm{cons}(1, \mathbf{nil})) \Rightarrow \\ (\emptyset \cup \emptyset \cup \{X = \mathbf{integer}; \mathbf{list}\,[X] = \mathbf{list}\,[Z]\}, \mathbf{list}\,[\mathrm{X}], \{X, Z\})\end{array}} \tag{4.25}$$

4.4 Most general unification

The following set of rules give a framework for a straightforward specification of a normalized most general unification algorithm.

The idea is basically to provide an equation system resolution based on the (type) variables and on the deconstructing of elaborate type forms. Doing so step by step in an equation system will lead to the establishment of the intended restrictions over the system variables, contained in the set given as the second argument for the mgu.

$$\frac{}{(\{\}, S) \Rightarrow \{\}} \ (\textit{Empty})$$

$$\frac{(\{E_1\{X \leftarrow W\}\{X \leftarrow W\}; \ldots; E_n\{X \leftarrow W\}\{X \leftarrow W\}\}, S \cup \{W\}) \Rightarrow \sigma}{(\{X = Y; E_1; \ldots; E_n\}, S) \Rightarrow \sigma \circ \{X \leftarrow W\} \circ \{Y \leftarrow W\}} \ (\textit{Variable link})$$
$(X, Y \in S)$
(W is a fresh name)

$$\frac{(\{E_1\{X \leftarrow T\}; \ldots; E_n\{X \leftarrow T\}\}, S) \Rightarrow \sigma}{(\{X = T; E_1; \ldots; E_n\}, S) \Rightarrow \sigma \circ \{X \leftarrow T\}} \ (\textit{Variable I})$$
$(X \in S)$

$$\frac{(\{E_1\{X \leftarrow T\}; \ldots; E_n\{X \leftarrow T\}\}, S) \Rightarrow \sigma}{(\{T = X; E_1; \ldots; E_n\}, S) \Rightarrow \sigma \circ \{X \leftarrow T\}} \ (\textit{Variable II})$$
$(X \in S)$

$$\frac{(\{X_1 = Y_1; \ldots; X_n = Y_n; E_1; \ldots; E_n\}, S) \Rightarrow \sigma}{(\{\mathrm{id}[X_1, \ldots, X_n] = \mathrm{id}[Y_1, \ldots, Y_n]; E_1; \ldots; E_n\}, S) \Rightarrow \sigma} \ (\textit{Parametric type resolution})$$

$$\frac{(\{U_1 = T_1; \ldots; U_n = T_n; U = T; E_1; \ldots; E_n\}, S) \Rightarrow \sigma}{(\{(U_1, \ldots, U_n)U = (T_1, \ldots, T_n)T; E_1; \ldots; E_n\}, S) \Rightarrow \sigma} \ (\textit{Constructor type resolution})$$

An example of the usage of this mgu algorithm specification is presented in (4.26), complementing the pattern verification presented in (4.17).

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{(\{\}, \{Z, V, Y_1, Y_2, W, W'\}) \Rightarrow \{\}}{(\{V = W\}, \{Z, V, Y_1, Y_2, W\}) \Rightarrow \{V \leftarrow W'\} \circ \{W \leftarrow W'\}}}{(\{\mathbf{list}\,[V] = \mathbf{list}\,[W]\}, \{Z, V, Y_1, Y_2, W\}) \Rightarrow \{V \leftarrow W'\} \circ \{W \leftarrow W'\}}}{(\{\mathbf{list}\,[W] = Y_2; \mathbf{list}\,[V] = \mathbf{list}\,[W]\}, \{Z, V, Y_1, Y_2, W\}) \Rightarrow \{V \leftarrow W'\} \circ \{W \leftarrow W'\} \circ \{Y_2 \leftarrow \mathbf{list}\,[W]\}}}{(\{Z = Y_1; \mathbf{list}\,[Z] = Y_2; \mathbf{list}\,[V] = \mathbf{list}\,[Z]\}, \{Z, V, Y_1, Y_2\}) \Rightarrow \{V \leftarrow W'\} \circ \{W \leftarrow W'\} \circ \{Y_2 \leftarrow \mathbf{list}\,[W]\} \circ \{Z \leftarrow W\} \circ \{Y_1 \leftarrow W\}} \tag{4.26}$$

4.5 Variable replacement

This section presents a possible implementation of the *replace* algorithm, which simply constructs, given a set of terms and a set of variable names, a new set of terms where each occurrence of a variable, contained in the argument set, is replaced by a fresh variable name. Along with this new set of terms the other two results of the algorithm are the set of new variables and a substitution that associates each new name with the name it replaced.

(*Variable in unitary set*)

$$\frac{}{replace(\{v_j\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{v'_1\}, \{v'_1\}, \{v'_1 \leftarrow v_j\})}$$

($j \in 1, \ldots, k$ and v'_1 is a fresh name)

(*Identifier in unitary set*)

$$\frac{}{replace(\{id\}, \{v_1, \ldots, v_k\}) \Rightarrow (\emptyset, \emptyset, \gamma)}$$

($\{id\} \cap \{v_1, \ldots, v_k\} = \emptyset$ and γ is the identity substitution)

(*Constructor in unitary set*)

$$\frac{replace(\{a_1, \ldots, a_n\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{a'_1, \ldots, a'_n\}, \{v'_1, \ldots, v'_m\}, \gamma)}{replace(\{id(a_1, \ldots, a_n)\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{id(a'_1, \ldots, a'_n)\}, \{v'_1, \ldots, v'_m\}, \gamma)}$$

(*Variable in n-ary set*)

$$\frac{replace(\{t_2, \ldots, t_w\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{t'_2, \ldots, t'_w\}, \{v'_1, \ldots, v'_m\}, \gamma)}{replace(\{v_j, t_2, \ldots, t_w\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{v'_{m+1}, t'_2, \ldots, t'_w\}, \{v'_1, \ldots, v'_m, v'_{m+1}\}, \gamma + \{v'_{m+1} \leftarrow v_j\})}$$

($j \in 1, \ldots, k$ and v'_{m+1} is a fresh name)

(*Identifier in n-ary set*)

$$\frac{replace(\{t_2, \ldots, t_w\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{t'_2, \ldots, t'_w\}, \{v'_1, \ldots, v'_m\}, \gamma)}{replace(\{id, t_2, \ldots, t_w\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{id, t'_2, \ldots, t'_w\}, \{v'_1, \ldots, v'_m\}, \gamma)}$$

($\{id\} \cap \{v_1, \ldots, v_k\} = \emptyset$)

(*Constructor in n-ary set*)

$$\frac{replace(\{a_1, \ldots, a_n, t_2, \ldots, t_w\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{a'_1, \ldots, a'_n, t'_2, \ldots, t'_w\}, \{v'_1, \ldots, v'_m\}, \gamma)}{replace(\{id(a_1, \ldots, a_n), t_2, \ldots, t_w\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{id(a'_1, \ldots, a'_n), t'_2, \ldots, t'_w\}, \{v'_1, \ldots, v'_m\}, \gamma)}$$

4.6 Implementation

The implementation of the type checker was made in a straightforward manner in respect to the specifications shown in this chapter. The language used in the development was Java [Jav] and, because Java elegantly supports subtyping, a Visitor scheme was used, consisting of no more than a case analysis on the class of the element being evaluated and therefore on the type being evaluated. This scheme, therefore, allows for an elegant treatment of all the abstract syntactic tree nodes, which are created by the parser. The evaluation starts when the whole program abstract syntactic tree is "visited". Appendix D shows the Visitor interface.

An essential element to the type check analysis is the environment where the declarations of both types and values are present, being the predefined types and values, such as **integer** and **nil**, introduced in the environment at creation time. To implement the environment hashtables were used, where the associations between names and their types or kinds and abbreviations are kept, depending if the name refers to a value or a kind, respectively.

To represent the scope of evaluation the environment holds a list of hashtables, being the current scope represented in the current cell of this list and the previous accessible by references. A scope is introduced when either a definition, a command or a restriction is being evaluated, which makes sense if one is to look to the scope of the declared variables in these constructions. Fresh scopes are also introduced when the evaluation is being made on a type or value declaration, through the language's top level declaration constructs, when there are type parameters involved, because it is very simple to introduce and remove these associations in the environment by introducing and then removing a fresh scope.

The considerations made in regard to the introduction of fresh scopes makes name hiding possible, being the only necessary condition the introduction of the same name in a different language construction. However, because the environment was developed in a sense to make it the most generic possible, the environment would support name hiding in the same level because it holds linked lists as the values of the associations, being the current value at the head and the shadowed values in the rest of the list, ordered correspondingly to their hiding order.

As can be noticed, no wheel was reinvented in the development of the type checker when it comes to the data structures involved, but considering the simplicity of the `POLY` constructs, regarding scope and other considerations that refer to the type checking, it makes sense that the data structures are also simple. There are, however, complex details that the type checker must resolve, as is for instance the message and the pattern type checking.

Because `POLY` offers polymorphic constructors it is necessary to evaluate messages that use these constructors in such a way that assures that their usage is being made coherently in respect to their declaration. In order to

```
declare op:[X](X)msg;
declare cons:[X](X, list[X])list[X];

op(cons(1, cons(2, nil))) // Correct
|
op(cons(1, cons(true, nil))) // Incorrect
```

Figure 4.13: Polymorphic constructor usage.

achieve this, recalling the *Constructor* rule, a correct substitution must be established for the type parameters involved, being the considered resolution of this problem, in the development of the POLY static checker, the usage of the mgu algorithm. Considering the equations that identify the types of the arguments used against the types present in the declaration of the constructor, it is possible to determine a valid substitution of the constructor's type parameters, and also, at top level, to determine if all type parameters have instantiations, allowing the type checker to conclude if a message construct is type grounded.

In the example shown in Figure 4.13 we can see a correct and an incorrect usage of a message construction. In the first case a substitution would be established for the type parameters, restricting them to integers, while in the second case no correct substitution is possible, because the type parameters can not be restricted to two different values. The mgu application to the equation system referred, will give the type checker the information needed to accept or reject message constructions in a very similar way as one of the steps required for pattern term evaluation.

So the mgu algorithm is essential to the type checking procedure, both in pattern terms and message constructions. The implementation of this algorithm is straightforward, as referred, from the specification provided previously, and basically consists in a case analysis on the type form involved in order to determine the correct rule to apply. Other than that, the algorithm needs only to keep track of the restrictions to complete it's task.

When a type error occurs, for instance due to a impossible resolution of an equation system by the mgu algorithm, an exception is launched, holding some error relative information. To help the POLY programmer optimize he's programming task, these exceptions do not cause the type checking procedure to fail immediately, being the evaluation point passed over to the next process, meaning the process that is in parallel composition with the process that caused the error to occur, allowing multiple errors to appear in just one static checker execution. This is also true in the top level declarations, where the next declarations will be evaluated (and not the process) if one of the declarations launches a type error exception. An example of a type error information is shown in Figure 4.14.

To finish off this comment on the implementation of the type checker

```
***
Found Type Error while evaluating agents pattern present in line 26, column 29
* Variable declarations and usage makes pattern typing incoherent *
* Context *

def <X> x, y:X; l:list[X] in op(x, l, y, l) [] inaction
|
***


***
Found Type Error while evaluating 'opInteger(true)' in line 30, column 4
* Term construction is typing incoherent *
* Context *
|
def opInteger(true) [] inaction
|
***
```

Figure 4.14: Type error information.

a small, but nevertheless important, remark must be made which is the usage, in the type checking procedure, of a unique fresh names generator, used for instance in the evaluation of constructors in order to consider their type parameters as unique identifiers, avoiding messy conclusions because of name interference. This name generator uses a special character that can not occur in any type scheme in order to assure that the created names are truly unique.

Chapter 5

Semantics

From an informal point of view the semantics of the language has been presented, starting in the example shown in section 2.1 and ending in the analysis of the type system presented in chapter 4. In order to provide the proper formalism of the language operational semantics, this chapter presents the reduction rules that illustrate system evolution.

5.1 Reduction rules

The most important reduction step in the `POLY` language is the activation of a definition (or a command, regarding their similarity). Other reduction steps involve the propagation of reductions through the restriction and parallel composition constructions. The following set of rules, extracted from [Cai99], illustrate the reductions present in the `POLY` language.

$$\frac{\Sigma, \mathrm{x} : T \vdash P \rightarrow \Sigma, \mathrm{x} : T \vdash Q}{\Sigma \vdash \textbf{new } \mathrm{x}\text{: } T \textbf{ in } P \rightarrow \Sigma \vdash \textbf{new } \mathrm{x}\text{: } T \textbf{ in } Q} \quad (\textit{Restriction reduction})$$

$$\frac{\Sigma \vdash P \rightarrow \Sigma \vdash Q}{\Sigma \vdash P \mid R \rightarrow \Sigma \vdash Q \mid R} \quad (\textit{Composition reduction})$$

$$\frac{\sigma : X \rightarrow \mathtt{T}, x \rightarrow \mathcal{T} \quad \sigma(p_i) = m_i \quad \Sigma \vdash \sigma(Q) \xrightarrow{*} \surd}{\begin{array}{c}\Sigma \vdash m^{i \in 1 \ldots n} \mid \textbf{def } <X>^{k \in 1 \ldots l} x : T^{j \in 1 \ldots d} \textbf{ in } p^{s \in 1 \ldots n} \ [Q] \ R \rightarrow \\ \Sigma \vdash \sigma(R) \mid \textbf{def } <X>^{k \in 1 \ldots l} x : T^{j \in 1 \ldots d} \textbf{ in } p^{s \in 1 \ldots n} \ [Q] \ R\end{array}} \quad (\textit{Definition reduction})$$

The first rule, identified by *Restriction reduction*, simply shows that if the continuation of a restriction has a possible reduction then the restriction will have itself a reduction, where the continuation is replaced by it's reduction.

Rule *Composition reduction* illustrates that if there is a reduction of one of the agents then the composition has the expected reduced form, consisting in the composition of the reduction and the other agent.

Rule *Definition reduction* shows that this reduction, as previously explained, is established when there are messages and a definition in the environment, such as the messages match the definition's acceptance pattern through a valid substitution in respect to the definition's value and type variables, and also if the test agent successfully reduces.

The successful reduction of the test agent involves launching it in an encapsulated environment and finding that, through a series of reductions, this environment is led to contain only persistent agents, namely definitions, and/or the inactive agent.

Both in the pattern matching procedure and in the test reduction an instantiation of variables may occur. In the first case the values used, are the ones that are present in the messages that matched the pattern in correspondence with the variable placement in the pattern terms. Also due to the matching procedure is the complete instantiation of type variables, if type variables are used. In the second case the values that make the test successful are the ones used to instantiate the respective variables, if any. Because of this, one can use the test agent in a declarative style, somewhat using unification as means of capturing values.

The test agent, along with it's declarative power, represents a guard for the activation of the definition, meaning that it holds a determined condition that must be verified. From a semantical viewpoint, this is the key feature of this definition (and command) component. The acceptance pattern is also capable of holding a condition, along with it's usage in providing input for the definition, through case analysis specified in the pattern constructors - but only case analysis base on value and not on type, once again, unless primitive values are used.

The activation of the definition implies the launching of the continuation agent in the environment, obviously affected by the instantiation of the variables, which are replaced with their respective values.

A final remark must be made, which is that both the command and the definition support everything stated up to this point, being the difference between these two `POLY` constructions the fact that the definition remains available after activation whilst the command does not. So the only difference that would exist in the rule would reside in the concluded reduced form, which, for the command rule, would only contain the continuation agent - regarding the referred substitution.

5.2 Initial environment

A `POLY` program is composed by a set of top level declarations, either of type or value nature, and a process, that can be considered as the body of the program. The reduction of the process is made through the steps explained in the previous section, being the environment considered the one created from the top level declarations, providing the rules shown in section 4.3.2 the basis of this creation.

Chapter 6

Subject reduction

This chapter concerns the usual result that one tries to achieve when developing a type system that basically refers to the ability of programs respecting type discipline while dynamically evolving. Since it is very technical one can always believe that the proof is well established and simply get the idea that if a determined process is well typed and this process evolves to another one, then this last one will be well typed. This gives us an interesting result because it allows us to say that no type errors will occur at runtime, particularly errors regarding the usage of type variables to confer polymorphism to the agent's acceptance pattern.

The first sections of the chapter involve simpler theorems that are proved separately just to facilitate reading, being established results used only in following sections. The last section contains the subject reduction theorem.

6.1 Value substitution in terms

Lemma 6.1.1 *If the judgements* $\Gamma, y : T \vdash t : V$ *and* $\Gamma \vdash m : T$ *are derivable, so is* $\Gamma \vdash t\{y \leftarrow m\} : V$.

Proof By induction on the structure of the typing derivation.

(**Case of Constructor**)
We have

$$\Gamma, y : T \vdash f(u_1, \ldots, u_n) : \sigma(U) \tag{6.1}$$

and

$$\Gamma \vdash m : T \tag{6.2}$$

(6.1) is concluded from:

$$\Gamma, y : T \vdash u_i : \sigma(T_i) \text{ (for } i = 1, \ldots, n) \tag{6.3}$$

and

$$\Gamma, y : T \vdash f : [\overline{X}](T_1, \ldots, T_n)U \tag{6.4}$$

By induction hypothesis on (6.3), we conclude

$$\Gamma \vdash u_i\{y \leftarrow m\} : \sigma(T_i) \text{ (for } i = 1, \ldots, n) \tag{6.5}$$

We must then consider two possible cases: either $y = f$ or $y \neq f$.

(*Case of* $y = f$)
We must have

$$T = [\overline{X}](T_1, \ldots, T_n)U$$

Also, (6.4) is of the form

$$\Gamma, f : [\overline{X}](T_1, \ldots, T_n)U \vdash f : [\overline{X}](T_1, \ldots, T_n)U$$

and (6.2) is of the form

$$\Gamma \vdash m : [\overline{X}](T_1, \ldots, T_n)U \tag{6.6}$$

(*Case of* $y \neq f$)
In this case, from (6.4) we immediately get

$$\Gamma \vdash f : [\overline{X}](T_1, \ldots, T_n)U \tag{6.7}$$

by weakening on y.

Either from (6.7) and (6.5), or (6.6) and (6.5), depending on whether $y \neq f$ or $y = f$, respectively, by an application of (Constructor), comes the intended conclusion

$$\Gamma \vdash f(u_1, \ldots, u_n)\{y \leftarrow m\} : \sigma(U) \tag{6.8}$$

$(f(u_1, \ldots, u_n)\{y \leftarrow m\} = f\{y \leftarrow m\}(u_1\{y \leftarrow m\}, \ldots, u_n\{y \leftarrow m\}))$

(**Case of Identifier**)
We have

$$\Gamma, y : T \vdash id : U \tag{6.9}$$

and

$$\Gamma \vdash m : T \tag{6.10}$$

We must consider two possible cases: either $y = id$ or $y \neq id$.

(*Case of* $y = id$)
We must have

$$T = U$$

Also, (6.9) is of the form

$$\Gamma, id : U \vdash id : U$$

and (6.10) is of the form

$$\Gamma \vdash m : U \tag{6.11}$$

(*Case of* $y \neq id$)
In this case, from (6.9) we immediately get

$$\Gamma \vdash id : U \tag{6.12}$$

by weakening on y.

Either from (6.11) or from (6.12), depending on whether $y = id$ or $y \neq id$, respectively, comes the intended conclusion

$$\Gamma \vdash id\{y \leftarrow m\} : U \tag{6.13}$$

(**Case of Value literal**)
We have

$$\Gamma, y : T \vdash v : V \tag{6.14}$$

and

$$\Gamma \vdash m : T \tag{6.15}$$

Regarding that $v \neq y$, the intended conclusion is obtained from (6.14) by weakening on y

$$\Gamma \vdash v\{y \leftarrow m\} : V \tag{6.16}$$

6.2 Value substitution in pattern terms

Lemma 6.2.1 *If the judgements* $\Gamma, w : U \vdash term(t) \Rightarrow (S, T, Z)$ *and* $\Gamma \vdash m : U$ *are derivable, so is* $\Gamma \vdash term(t\{w \leftarrow m\}) \Rightarrow (S, T, Z)$.

Proof By induction on the structure of the typing derivation.

(**Case of Pattern identifier**)
We have

$$\Gamma, w : U \vdash term(id) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.17}$$

and

$$\Gamma \vdash m : U \tag{6.18}$$

We must consider two possible cases: either $id = w$ or $id \neq w$.

(*Case of* $id = w$)
We must have

$$[\overline{Z}]T = U$$

Also, (6.17) is of the form

$$\Gamma, id : [\overline{Z}]T \vdash term(id) \Rightarrow (\emptyset, T, \{\overline{Z}\})$$

and (6.18) is of the form

$$\Gamma \vdash m : [\overline{Z}]T$$

so $m : [\overline{Z}]T \in \Gamma$ which, by application of the (Pattern identifier) gives us

$$\Gamma \vdash term(m) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.19}$$

(*Case of* $id \neq w$)
In this case, from (6.17) we immediately get

$$\Gamma \vdash term(id) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.20}$$

by weakening on w.

Either from (6.19) or (6.20), depending on whether $id = w$ or $id \neq w$, respectively, comes the intended conclusion

$$\Gamma \vdash term(id\{w \leftarrow m\}) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.21}$$

(**Case of Pattern value literal**)
We have

$$\Gamma, w : U \vdash term(v) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.22}$$

and

$$\Gamma \vdash m : U \tag{6.23}$$

Regarding that $v \neq w$, the intended conclusion is obtained from (6.22) by weakening on y

$$\Gamma \vdash term(v\{w \leftarrow m\}) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.24}$$

(**Case of Pattern constructor**)
We have

$$\begin{gathered}\Gamma, w : U \vdash term(id(u_1, \ldots, u_n)) \\ \Rightarrow (S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, T, \{\overline{Z}\} \cup V_1 \cup \ldots \cup V_n)\end{gathered} \tag{6.25}$$

and

$$\Gamma \vdash m : U \tag{6.26}$$

(6.25) is concluded from

$$\Gamma, w : U \vdash term(u_i) \Rightarrow (S_i, U_i, V_i) \text{ (for } i = 1, \ldots, n) \tag{6.27}$$

and

$$\Gamma, w : U \vdash id : [\overline{Z}](T_1, \ldots, T_n)T \tag{6.28}$$

by induction hypothesis on (6.27), we conclude

$$\Gamma \vdash term(u_i\{w \leftarrow m\}) \Rightarrow (S_i, U_i, V_i) \text{ (for } i = 1, \ldots, n) \tag{6.29}$$

We must then consider two possible cases: either $id = w$ or $id \neq w$.

(*Case of* $id = w$)
We must have

$$U = [\overline{Z}](T_1, \ldots, T_n)T$$

Also, (6.28) is of the form

$$\Gamma, id : [\overline{Z}](T_1, \ldots, T_n)T \vdash id : [\overline{Z}](T_1, \ldots, T_n)T$$

And (6.26) is of the form

$$\Gamma \vdash m : [\overline{Z}](T_1, \ldots, T_n)T \tag{6.30}$$

(*Case of* $id \neq w$)
In this case, from (6.28) we immediately get

$$\Gamma \vdash id : [\overline{Z}](T_1, \ldots, T_n)T \tag{6.31}$$

by weakening on w.

Either from (6.30) and (6.29), or (6.31) and (6.29), depending on whether $id = w$ or $id \neq w$, respectively, by an application of (Pattern constructor), comes the intended conclusion

$$\begin{gathered}\Gamma \vdash term(id(u_1, \ldots, u_n)\{w \leftarrow m\}) \\ \Rightarrow (S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, T, \{\overline{Z}\} \cup V_1 \cup \ldots \cup V_n) \\ (id(u_1, \ldots, u_n)\{w \leftarrow m\} \equiv id\{w \leftarrow m\}(u_1\{w \leftarrow m\}, \ldots, u_n\{w \leftarrow m\}))\end{gathered} \tag{6.32}$$

6.3 Value substitution in processes

Lemma 6.3.1 *If the judgements* $\Gamma, w : U \vdash P \quad OK$ *and* $\Gamma \vdash m : U$ *are derivable, so is* $\Gamma \vdash P\{w \leftarrow m\} \quad OK$.

Proof By induction on the structure of the typing derivation.

(**Case of Definition**)
We have

$$\Gamma, w : U \vdash \mathbf{def} < \overline{X} > v_1 : T_1; \ldots; v_k : T_k \ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q \quad OK \tag{6.33}$$

and

$$\Gamma \vdash m : U \tag{6.34}$$

(6.33) is concluded from

$$replace(\{M_1, \ldots, M_t\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{M'_1, \ldots, M'_t\}, \{v'_1, \ldots, v'_m\}, \gamma) \tag{6.35}$$

and

$$\Gamma, w : U, \overline{Y :: 0}, v'_1 : Y_1, \ldots, v'_m : Y_m \vdash term(M'_i) \Rightarrow (S_i, \mathbf{msg}, Z_i) \quad (\text{for } i = 1, \ldots, t) \tag{6.36}$$

and

$$\sigma = mgu_n(S_1 \cup \ldots \cup S_t, \{\overline{Y}\} \cup Z_1 \cup \ldots \cup Z_t) \tag{6.37}$$

and

$$\varphi = mgu_n(\bigcup_{q : Y_q \in dom(\sigma) \wedge \exists_l . \gamma(v'_q) = v_l} \{T_l = \sigma(Y_q)\}, \{\overline{X}\}) \tag{6.38}$$

and

$$\Gamma, w : U, \overline{X :: 0}, v_1 : T_1, \ldots, v_k : T_k \vdash P \quad OK \tag{6.39}$$

and

$$\Gamma, w : U, \overline{X :: 0}, v_1 : T_1, \ldots, v_k : T_k \vdash Q \quad OK \tag{6.40}$$

From (6.35), knowing that for all $i \in 1, \ldots, k$ $v_k \neq w$, we conclude

$$\begin{array}{c} replace(\{M_1\{w \leftarrow m\}, \ldots, M_t\{w \leftarrow m\}\}, \{v_1, \ldots, v_k\}) \Rightarrow \\ (\{M'_1\{w \leftarrow m\}, \ldots, M'_t\{w \leftarrow m\}\}, \{v'_1, \ldots, v'_m\}, \gamma) \end{array} \tag{6.41}$$

Also, we know that

$$\Gamma, w : U, \overline{Y :: 0}, v'_1 : Y_1, \ldots, v'_m : Y_m \equiv \Gamma, \overline{Y :: 0}, v'_1 : Y_1, \ldots, v'_m : Y_m, w : U \tag{6.42}$$

and that

$$\Gamma, w : U, \overline{X :: 0}, v_1 : T_1, \ldots, v_k : T_k \equiv \Gamma, \overline{X :: 0}, v_1 : T_1, \ldots, v_k : T_k, w : U \tag{6.43}$$

By Lemma 6.2.1, having (6.36) and (6.34) and considering (6.42), we conclude

$$\Gamma, \overline{Y :: 0}, v'_1 : Y_1, \ldots, v'_m : Y_m \vdash term(M'_i\{w \leftarrow m\}) \Rightarrow (S_i, \mathbf{msg}, Z_i) \quad (\text{for } i = 1, \ldots, t) \tag{6.44}$$

By induction hypothesis on (6.39), considering (6.43), we conclude

$$\Gamma, \overline{X :: 0}, v_1 : T_1, \ldots, v_k : T_k \vdash P\{w \leftarrow m\} \quad OK \tag{6.45}$$

By induction hypothesis on (6.40), considering (6.43), we conclude

$$\Gamma, \overline{X :: 0}, v_1 : T_1, \ldots, v_k : T_k \vdash Q\{w \leftarrow m\} \quad OK \tag{6.46}$$

From (6.41), (6.44), (6.37), (6.38), (6.45) and (6.46), by an application of (Definition), comes the intended conclusion

$$\Gamma \vdash (\mathbf{def} < \overline{X} > \overline{v : T} \ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q)\{w \leftarrow m\}\} \quad OK \tag{6.47}$$

$$\begin{gathered}((\mathbf{def} < \overline{X} > \overline{v : T} \ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q)\{w \leftarrow m\} \\ \equiv \\ \mathbf{def} < \overline{X} > \overline{v : T} \ \mathbf{in}\ M_1\{w \leftarrow m\} \& \ldots \& M_t\{w \leftarrow m\} \\ [P\{w \leftarrow m\}]Q\{w \leftarrow m\})\end{gathered}$$

(**Case of Message**)
We have

$$\Gamma, w : U \vdash M \quad OK \tag{6.48}$$

and

$$\Gamma \vdash m : U \tag{6.49}$$

(6.48) is concluded from

$$\Gamma, w : U \vdash M : \mathbf{msg} \tag{6.50}$$

By Lemma 6.1.1, having (6.50) and (6.49), we conclude

$$\Gamma \vdash M\{w \leftarrow m\} : \mathbf{msg} \tag{6.51}$$

From (6.51), by an application of (Message), comes the intended conclusion

$$\Gamma \vdash M\{w \leftarrow m\} \quad OK \tag{6.52}$$

(**Case of Restriction**)
We have

$$\Gamma, w : U \vdash \mathbf{new}\ v_1 : T_1; \ldots; v_n : T_n \ \mathbf{in}\ P \quad OK \tag{6.53}$$

and

$$\Gamma \vdash m : U \tag{6.54}$$

(6.53) is concluded from

$$\Gamma, w : U, v_1 : T_1, \ldots, v_n : T_n \vdash P \quad OK \tag{6.55}$$

Also, we know that

$$\Gamma, w : U, v_1 : T_1, \ldots, v_n : T_n \equiv \Gamma, v_1 : T_1, \ldots, v_n : T_n, w : U \tag{6.56}$$

By induction hypothesis on (6.55), considering (6.56), we conclude

$$\Gamma, v_1 : T_1, \ldots, v_n : T_n \vdash P\{w \leftarrow m\} \quad OK \tag{6.57}$$

From (6.57), by an application of (Restriction), comes the intended conclusion

$$\Gamma \vdash (\textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P)\{w \leftarrow m\} \quad OK \tag{6.58}$$

$$((\textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P)\{w \leftarrow m\}$$
$$\equiv$$
$$\textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P\{w \leftarrow m\})$$

(**Case of Parallel composition**)
We have

$$\Gamma, w : U \vdash P \mid Q \quad OK \tag{6.59}$$

and

$$\Gamma \vdash m : U \tag{6.60}$$

(6.59) is concluded from

$$\Gamma, w : U \vdash P \quad OK \tag{6.61}$$

and

$$\Gamma, w : U \vdash Q \quad OK \tag{6.62}$$

By induction hypothesis on (6.61), we conclude

$$\Gamma \vdash P\{w \leftarrow m\} \quad OK \tag{6.63}$$

By induction hypothesis on (6.62), we conclude

$$\Gamma \vdash Q\{w \leftarrow m\} \quad OK \tag{6.64}$$

From (6.63) and (6.64), by an application of (Parallel composition), comes the intended conclusion

$$\Gamma \vdash (P \mid Q)\{w \leftarrow m\} \quad OK \tag{6.65}$$

$$((P \mid Q)\{w \leftarrow m\} \equiv P\{w \leftarrow m\} \mid Q\{w \leftarrow m\})$$

(**Case of Inaction**)
We have

$$\Gamma, w : U \vdash \mathbf{inaction} \quad OK \tag{6.66}$$

and

$$\Gamma \vdash m : U \tag{6.67}$$

Since $\mathbf{inaction}\{w \leftarrow m\} \equiv \mathbf{inaction}$, the conclusion is obtained by an application of (Inaction)

$$\Gamma \vdash \mathbf{inaction}\{w \leftarrow m\} \quad OK \tag{6.68}$$

6.4 Type substitution in terms

Lemma 6.4.1 *If γ and σ are substitutions over $\overline{Y}$, being $\gamma(Y_i) \doteq \sigma(Y_i)\{X \leftarrow U\}$, having $\{\overline{Y}\} \cap (vars(U) \cup \{X\}) = \emptyset$, then $\{X \leftarrow U\} \circ \sigma(T) = \gamma(T\{X \leftarrow U\})$*

Proof The equivalence is straightforward, regarding the $\{\overline{Y}\} \cap (vars(U) \cup \{X\}) = \emptyset$ precondition.

$$\{X \leftarrow U\} \circ \sigma(T) = \{X \leftarrow U\} \circ \sigma(T\{X \leftarrow U\}) = \gamma(T\{X \leftarrow U\}) \quad (6.69)$$

Lemma 6.4.2 *If the judgements $\Gamma, X :: 0, \Gamma' \vdash t : V$ and $\Gamma \vdash U :: 0$ are derivable, so is $\Gamma, \Gamma'\{X \leftarrow U\} \vdash t : V\{X \leftarrow U\}$.*

Proof By induction on the structure of the typing derivation.

(**Case of Constructor**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash f(a_1, \ldots, a_n) : \sigma(T) \quad (6.70)$$

and

$$\Gamma \vdash U :: 0 \quad (6.71)$$

(6.70) is concluded from:

$$\Gamma, X :: 0, \Gamma' \vdash a_i : \sigma(T_i) \text{ (for } i = 1, \ldots, n) \quad (6.72)$$

and

$$\Gamma, X :: 0, \Gamma' \vdash f : [\overline{Y}](T_1, \ldots, T_n)T \quad (6.73)$$

By induction hypothesis on (6.72), we conclude

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash a_i : \sigma(T_i)\{X \leftarrow U\} \text{ (for } i = 1, \ldots, n) \quad (6.74)$$

Considering $\gamma(Y_i) \doteq \{X \leftarrow U\} \circ \sigma(Y_i)$, regarding that $\sigma(T_i)\{X \leftarrow U\} \equiv \{X \leftarrow U\} \circ \sigma(T_i)$ and that $\{\overline{Y}\} \cap (vars(U) \cup \{X\}) = \emptyset$, by Lemma 6.4.1, from (6.74) we get

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash a_i : \gamma(T_i\{X \leftarrow U\}) \text{ (for } i = 1, \ldots, n) \quad (6.75)$$

From (6.73), considering (6.71) and that $(vars(U) \cap \{\overline{Y}\} = \emptyset)$, we conclude

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash f : [\overline{Y}](T_1\{X \leftarrow U\}, \ldots, T_n\{X \leftarrow U\})T\{X \leftarrow U\} \quad (6.76)$$

From (6.76) and (6.75), by an application of (Constructor), we obtain

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash f(a_1, \ldots, a_n) : \gamma(T\{X \leftarrow U\}) \quad (6.77)$$

From (6.77), once again by Lemma 6.4.1, comes the intended conclusion

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash f(a_1, \ldots, a_n) : \sigma(T)\{X \leftarrow U\} \quad (6.78)$$
$$(\sigma(T)\{X \leftarrow U\} \equiv \{X \leftarrow U\} \circ \sigma(T))$$

(**Case of Identifier**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash id : T \tag{6.79}$$

and

$$\Gamma \vdash U :: 0 \tag{6.80}$$

The intended conclusion is straightforward from (6.79) and (6.80)

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash id : T\{X \leftarrow U\} \tag{6.81}$$

(**Case of Value literal**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash v : V \tag{6.82}$$

and

$$\Gamma \vdash U :: 0 \tag{6.83}$$

The intended conclusion, due to $V\{X \leftarrow U\} = V$, is straightforward from (6.82)

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash v : V\{X \leftarrow U\} \tag{6.84}$$

6.5 Type substitution in pattern terms

Lemma 6.5.1 *If the judgements* $\Gamma, X :: 0, \Gamma' \vdash term(t) \Rightarrow (S, T, Z)$ *and* $\Gamma \vdash U :: 0$ *are derivable, so is* $\Gamma, \Gamma'\{X \leftarrow U\} \vdash term(t) \Rightarrow (S, T, Z)\{X \leftarrow U\}$.

Proof By induction on the structure of the typing derivation.

(**Case of Pattern constructor**)
We have

$$\begin{array}{c}\Gamma, X :: 0, \Gamma' \vdash term(id(u_1, \ldots, u_n)) \\ \Rightarrow (S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, T, \{\overline{Z}\} \cup V_1 \cup \ldots \cup V_n)\end{array} \tag{6.85}$$

and

$$\Gamma \vdash U :: 0 \tag{6.86}$$

(6.85) is concluded from

$$\Gamma, X :: 0, \Gamma' \vdash term(u_i) \Rightarrow (S_i, U_i, V_i) \text{ (for } i = 1, \ldots, n) \tag{6.87}$$

and

$$\Gamma, X :: 0, \Gamma' \vdash id : [\overline{Z}](T_1, \ldots, T_n)T \tag{6.88}$$

By induction hypothesis on (6.87), we conclude

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash term(u_i) \Rightarrow (S_i, U_i, V_i)\{X \leftarrow U\} \text{ (for } i = 1, \ldots, n) \tag{6.89}$$

$$((S_i, U_i, V_i)\{X \leftarrow U\} \equiv (S_i\{X \leftarrow U\}, U_i\{X \leftarrow U\}, V_i)$$

From (6.88), regarding (6.86) and $\{\overline{Z}\} \cap \{X\} = \emptyset$, we obtain

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash id : [\overline{Z}](T_1\{X \leftarrow U\}, \ldots, T_n\{X \leftarrow U\})T\{X \leftarrow U\} \tag{6.90}$$

From (6.90) and (6.89), by an application of (Pattern constructor), comes the intended conclusion

$$\begin{array}{c}\Gamma, \Gamma'\{X \leftarrow U\} \vdash term(id(u_1, \ldots, u_n)) \\ \Rightarrow (S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, T, \\ \{\overline{Z}\} \cup V_1 \cup \ldots \cup V_n)\{X \leftarrow U\}\end{array} \tag{6.91}$$

$$\begin{array}{c}((S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, T, \\ \{\overline{Z}\} \cup V_1 \cup \ldots \cup V_n)\{X \leftarrow U\} \\ \equiv \\ (S_1\{X \leftarrow U\} \cup \ldots \cup S_n\{X \leftarrow U\} \cup \{T_1\{X \leftarrow U\} = U_1\{X \leftarrow U\}; \ldots; \\ T_n\{X \leftarrow U\} = U_n\{X \leftarrow U\}\}, T\{X \leftarrow U\}, \{\overline{Z}\} \cup V_1 \cup \ldots \cup V_n))\end{array}$$

(**Case of Pattern identifier**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash term(id) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.92}$$

and

$$\Gamma \vdash U :: 0 \tag{6.93}$$

From (6.92) and (6.93) comes the intended conclusion

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash term(id) \Rightarrow (\emptyset, T, \{\overline{Z}\})\{X \leftarrow U\} \tag{6.94}$$

$$((\emptyset, T, \{\overline{Z}\})\{X \leftarrow U\} \equiv (\emptyset, T\{X \leftarrow U\}, \{\overline{Z}\})$$

(**Case of Pattern value literal**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash term(v) \Rightarrow (\emptyset, T, \{\overline{Z}\}) \tag{6.95}$$

and

$$\Gamma \vdash U :: 0 \tag{6.96}$$

Regarding that $T\{X \leftarrow U\} = T$, from (6.95) comes the intended conclusion

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash term(v) \Rightarrow (\emptyset, T, \{\overline{Z}\})\{X \leftarrow U\} \tag{6.97}$$

6.6 Type substitution in processes

Lemma 6.6.1 *If σ and γ are two substitutions such that $dom(\sigma) \cap Im(\gamma) = \emptyset$ and $dom(\sigma) \cap dom(\gamma) = \emptyset$, then $\sigma \circ \gamma \equiv \sigma + \gamma$.*

Proof If σ does not act upon any element introduced by γ and the domains are independent then the substitutions are also independent, and can be written as so.

Definition 6.6.2 *A substitution γ is the most general unification (mgu) of an equation system S if γ is an unifier, meaning that $\forall (T = T') \in S \ \gamma(T) = \gamma(T')$, and if it is most general, meaning that $\forall \gamma'$ unifier of $S \ \exists \sigma : \gamma' = \sigma \circ \gamma$.*

Lemma 6.6.3 *If γ is mgu of S, having $\{\overline{Y}\}$ as domain, for all X and U such that $\{X\} \cap \{\overline{Y}\} = \emptyset$ and $vars(U) = \emptyset$, then γ' is mgu of $S\{X \leftarrow U\}$, where $\gamma' = \{X \leftarrow U\} \circ \gamma$.*

Proof Regarding the definition of mgu it is necessary to prove that γ' is unifier and that it is most general.
(**Unifier**)
Because γ is mgu of S we have that

$$\forall (T = T') \in S \quad \gamma(T) = \gamma(T') \tag{6.98}$$

From (6.98) we can write

$$\forall (T = T') \in S \quad \{X \leftarrow U\} \circ \gamma(T) = \{X \leftarrow U\} \circ \gamma(T') \tag{6.99}$$

Because $\{X\} \cap \{\overline{Y}\} = \emptyset$ and $vars(U) \cap \{\overline{Y}\} = \emptyset$, recalling that $dom(\gamma) = \{\overline{Y}\}$, from (6.99) we conclude

$$\forall (T = T') \in S \quad \{X \leftarrow U\} \circ \gamma(T\{X \leftarrow U\}) = \{X \leftarrow U\} \circ \gamma(T'\{X \leftarrow U\}) \tag{6.100}$$

Recalling that $\gamma' = \{X \leftarrow U\} \circ \gamma$, from (6.100) we get that

$$\forall (T = T') \in S \quad \gamma'(T\{X \leftarrow U\}) = \gamma'(T'\{X \leftarrow U\}) \tag{6.101}$$

Regarding that for all $(W = W') \in S\{X \leftarrow U\}$ there is $(T = T') \in S$ such that $W = T\{X \leftarrow U\}$ and $W' = T'\{X \leftarrow U\}$, we conclude

$$\forall (W = W') \in S\{X \leftarrow U\} \quad \gamma'(W) = \gamma'(W') \tag{6.102}$$

(**Most general**)
Considering γ'' an unifier of $S\{X \leftarrow U\}$

$$\forall (W = W') \in S\{X \leftarrow U\} \quad \gamma''(W) = \gamma''(W') \tag{6.103}$$

Regarding that if $(W = W') \in S\{X \leftarrow U\}$ then there is $T = T' \in S$ such that $W = T\{X \leftarrow U\}$ and $W' = T'\{X \leftarrow U\}$, we conclude

$$\forall (T = T') \in S \quad \gamma''(T\{X \leftarrow U\}) = \gamma''(T'\{X \leftarrow U\}) \tag{6.104}$$

(6.104) can be written as

$$\forall (T = T') \in S \quad \gamma'' \circ \{X \leftarrow U\}(T) = \gamma'' \circ \{X \leftarrow U\}(T') \tag{6.105}$$

So, from (6.105) we conclude that $\gamma'' \circ \{X \leftarrow U\}$ is unifier of S, and knowing that γ is mgu of S we get that

$$\exists \sigma : \gamma'' \circ \{X \leftarrow U\} = \sigma \circ \gamma \tag{6.106}$$

Knowing that $X \in dom(\gamma'' \circ \{X \leftarrow U\})$ we conclude that $X \in dom(\sigma \circ \gamma)$ but since $dom(\gamma) = \{\overline{Y}\}$ we get that $X \in dom(\sigma)$, which allows us to write (6.106) as

$$\gamma'' \circ \{X \leftarrow U\} = (\sigma\lfloor_X + \sigma\lfloor_{\overline{X}}) \circ \gamma \tag{6.107}$$

We can also establish a sum from $\gamma'' \circ \{X \leftarrow U\}$ in regard to it's domain, considering the partition obtained due to X, which, considering (6.107), gives us

$$\gamma''\lfloor_{\overline{X}} + \gamma''\lfloor_{\{vars(U)\}} \circ \{X \leftarrow U\} = (\sigma\lfloor_X + \sigma\lfloor_{\overline{X}}) \circ \gamma \tag{6.108}$$

Looking at the substitution on the left hand side of the equation we conclude that $\{X\} \cap Im(\gamma''\lfloor_{\overline{X}} + \gamma''\lfloor_{\{vars(U)\}} \circ \{X \leftarrow U\}) = \emptyset$, because $\{X\} \cap vars(S\{X \leftarrow U\}) = \emptyset$, which allows us to conclude that $\{X\} \cap Im(\sigma\lfloor_X) = \emptyset$ and that $\{X\} \cap Im(\sigma\lfloor_{\overline{X}}) = \emptyset$, which along with $dom(\sigma\lfloor_X) = \{X\}$ and $\{X\} \cap dom(\sigma\lfloor_{\overline{X}}) = \emptyset$, by Lemma 6.6.1, from (6.108) we conclude

$$\gamma''\lfloor_{\overline{X}} + \gamma''\lfloor_{\{vars(U)\}} \circ \{X \leftarrow U\} = (\sigma\lfloor_X \circ \sigma\lfloor_{\overline{X}}) \circ \gamma \tag{6.109}$$

Considering now the domain partition obtained from considering the sum established from γ in regard to X in (6.109), we obtain

$$\gamma''\lfloor_{\overline{X}} + \gamma''\lfloor_{\{vars(U)\}} \circ \{X \leftarrow U\} = \sigma\lfloor_X \circ \sigma\lfloor_{\overline{X}} \circ \gamma\lfloor_X + \sigma\lfloor_X \circ \sigma\lfloor_{\overline{X}} \circ \gamma\lfloor_{\overline{X}} \tag{6.110}$$

Restricting both sides of (6.110), discarding $\overline{X}$, we obtain

$$\gamma''\lfloor_{\{vars(U)\}} \circ \{X \leftarrow U\} = \sigma\lfloor_X \circ \sigma\lfloor_{\overline{X}} \circ \gamma\lfloor_X \tag{6.111}$$

From (6.111), regarding that $\{X\} \cap dom(\gamma) = \emptyset$ and $\{X\} \cap dom(\sigma\lfloor_{\overline{X}}) = \emptyset$, we conclude

$$\sigma\lfloor_X = \gamma''\lfloor_{\{vars(U)\}} \circ \{X \leftarrow U\} \tag{6.112}$$

Restricting both sides of (6.110), discarding X, we obtain

$$\gamma''\lfloor_{\overline{X}} = \sigma\lfloor_X \circ \sigma\lfloor_{\overline{X}} \circ \gamma\lfloor_{\overline{X}} \tag{6.113}$$

From (6.113) and (6.112) we conclude

$$\gamma''\lfloor_{\overline{X}} = \gamma''\lfloor_{\{vars(U)\}} \circ \{X \leftarrow U\} \circ \sigma\lfloor_{\overline{X}} \circ \gamma\lfloor_{\overline{X}} \tag{6.114}$$

Recalling that $\{X\} \cap Im(\sigma\lfloor_{\overline{X}}) = \emptyset$ and knowing that $Im(\{X \leftarrow U\}) = \emptyset$, since $vars(U) = \emptyset$, from (6.114) we conclude

$$\gamma''\lfloor_{\overline{X}} = \gamma''\lfloor_{\{vars(U)\}} \circ \sigma\lfloor_{\overline{X}} \circ \{X \leftarrow U\} \circ \gamma\lfloor_{\overline{X}} \tag{6.115}$$

Since from $\{X\} \cap dom(\gamma'') = \emptyset$ we get that $\gamma''\lfloor_{\overline{X}} = \gamma''$ and from $dom(\gamma) = \{\overline{Y}\}$ we get that $\gamma\lfloor_{\overline{X}} = \gamma$, recalling that $\gamma' = \{X \leftarrow U\} \circ \gamma$ and considering substitution $\psi = \gamma''\lfloor_{\{vars(U)\}} \circ \sigma\lfloor_{\overline{X}}$, from (6.115) we obtain the intended conclusion

$$\gamma'' = \psi \circ \gamma' \tag{6.116}$$

From (6.102) and (6.116), regarding Definition 6.6.2, we conclude that γ' is mgu of $S\{X \leftarrow U\}$.

Lemma 6.6.4 *If the judgements* $\Gamma, X :: 0, \Gamma' \vdash P \quad OK$ *and* $\Gamma \vdash U :: 0$ *are derivable, so is* $\Gamma, \Gamma'\{X \leftarrow U\} \vdash P\{X \leftarrow U\} \quad OK$.

Proof By induction on the structure of the typing derivation.

(**Case of Definition**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash \mathbf{def} < \overline{Z} > v_1 : T_1; \ldots; v_k : T_k \ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q \quad OK \tag{6.117}$$

and

$$\Gamma \vdash U :: 0 \tag{6.118}$$

(6.117) is concluded from

$$replace(\{M_1, \ldots, M_t\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{M'_1, \ldots, M'_t\}, \{v'_1, \ldots, v'_m\}, \gamma) \tag{6.119}$$

and

$$\Gamma, X :: 0, \Gamma', \overline{Y :: 0}, v'_1 : Y_1, \ldots, v'_m : Y_m \vdash term(M'_i) \Rightarrow (S_i, \mathbf{msg}, R_i) \quad (\text{for } i = 1, \ldots, t) \tag{6.120}$$

and

$$\sigma = mgu_n(S_1 \cup \ldots \cup S_t, \{\overline{Y}\} \cup R_1 \cup \ldots \cup R_t) \tag{6.121}$$

and

$$\varphi = mgu_n(\bigcup_{q:Y_q \in dom(\sigma) \wedge \exists_l . \gamma(v'_q) = v_l} \{T_l = \sigma(Y_q)\}, \{\overline{Z}\}) \tag{6.122}$$

$$\begin{array}{l}(\forall_{i \in 1,\ldots,h}, Z_i \in \{\overline{Z}\}\ \exists_j : \varphi(Z_i) = W_j) \\ (\forall_{i,j \in 1,\ldots,h} : Z_i, Z_j \in \{\overline{Z}\}\ \varphi(Z_i) = \varphi(Z_j) \Rightarrow i = j)\end{array} \tag{6.123}$$

and

$$\Gamma, X :: 0, \Gamma', \overline{Z :: 0}, v_1 : T_1, \ldots, v_k : T_k \vdash P \quad OK \tag{6.124}$$

and

$$\Gamma, X :: 0, \Gamma', \overline{Z :: 0}, v_1 : T_1, \ldots, v_k : T_k \vdash Q \quad OK \tag{6.125}$$

Regarding that $\{X\} \cap \{\overline{Y}\} = \emptyset$ we have

$$\forall_{i \in 1,\ldots,k}\ Y_i\{X \leftarrow U\} = Y_i \tag{6.126}$$

and noticing that $\{X\} \cap \{\overline{Z}\} = \emptyset$ we also have

$$\forall_{i \in 1,\ldots,h}, Z_i \in \{\overline{Z}\} : Z_i\{X \leftarrow U\} = Z_i \tag{6.127}$$

By Lemma 6.5.1, having (6.120) and (6.118), considering (6.126), we obtain

$$\begin{array}{c}\Gamma, \Gamma'\{X \leftarrow U\}, \overline{Y :: 0}, v'_1 : Y_1, \ldots, v'_m : Y_m \vdash term(M'_i) \Rightarrow \\ (S_i, \mathbf{msg}, R_i)\{X \leftarrow U\}\ (\text{for } i = 1, \ldots, t)\end{array} \tag{6.128}$$

$$((S_i, \mathbf{msg}, R_i)\{X \leftarrow U\} \equiv (S_i\{X \leftarrow U\}, \mathbf{msg}, R_i))$$

By induction hypothesis on (6.124), considering (6.127), we conclude

$$\begin{array}{c}\Gamma, \Gamma'\{X \leftarrow U\}, \overline{Z :: 0}, v_1 : (T_1\{X \leftarrow U\}), \ldots, v_k : (T_k\{X \leftarrow U\}) \\ \vdash P\{X \leftarrow U\} \quad OK\end{array} \tag{6.129}$$

By induction hypothesis on (6.125), considering (6.127), we conclude

$$\begin{array}{c}\Gamma, \Gamma'\{X \leftarrow U\}, \overline{Z :: 0}, v_1 : (T_1\{X \leftarrow U\}), \ldots, v_k : (T_k\{X \leftarrow U\}) \\ \vdash Q\{X \leftarrow U\} \quad OK\end{array} \tag{6.130}$$

From (6.121), regarding that $vars(U) = \emptyset$, concluded from (6.118), and that $\{X\} \cap (\{\overline{Y}\} \cup R_1 \cup \ldots \cup R_t) = \emptyset$, by Lemma 6.6.3 we conclude

$$\{X \leftarrow U\} \circ \sigma = mgu_n((S_1 \cup \ldots \cup S_t)\{X \leftarrow U\}, \{\overline{Y}\} \cup R_1 \cup \ldots \cup R_t) \tag{6.131}$$

$$((S_1 \cup \ldots \cup S_t)\{X \leftarrow U\} \equiv (S_1\{X \leftarrow U\} \cup \ldots \cup S_t\{X \leftarrow U\}))$$

We must know consider the new equation system

$$\bigcup_{q:Y_q \in dom(\sigma) \wedge \exists_l . \gamma(v'_q) = v_l} \{(T_l\{X \leftarrow U\}) = \{X \leftarrow U\} \circ \sigma(Y_q)\} \tag{6.132}$$

$$\equiv (\bigcup_{q:Y_q \in dom(\sigma_l) \wedge \exists_l . \gamma(v'_q) = v_l} \{T_l = \sigma(Y_q)\})\{X \leftarrow U\}$$

From (6.122), noticing that $dom(\varphi) = \{\overline{Z}\}$, having $\{X\} \cap \{\overline{Z}\} = \emptyset$ and, from (6.118), $vars(U) = \emptyset$, by Lemma 6.6.3, $\{X \leftarrow U\} \circ \varphi$ is mgu of (6.132). From (6.123) we know that

$$\begin{array}{l} \forall_{i \in 1,\ldots,h}, Z_i \in \{\overline{Z}\}\ \exists_j : \varphi(Z_i) = W_j \\ \forall_{i,j \in 1,\ldots,h} : Z_i, Z_j \in \{\overline{Z}\}\ \varphi(Z_i) = \varphi(Z_j) \Rightarrow i = j \end{array} \tag{6.133}$$

From (6.133), noticing that $\forall_j W_j\{X \leftarrow U\} = W_j$ we conclude

$$\forall_{i \in 1,\ldots,h}, Z_i \in \{\overline{Z}\}\ \exists_j : \{X \leftarrow U\} \circ \varphi(Z_i) = W_j \tag{6.134}$$

Also from (6.133), noticing that $\forall_i \{X \leftarrow U\} \circ \varphi(Z_i) = \varphi(Z_i)$, we get that $\{X \leftarrow U\} \circ \varphi(Z_i) = \{X \leftarrow U\} \circ \varphi(Z_j) \equiv \varphi(Z_i) = \varphi(Z_j)$, so

$$\forall_{i,j \in 1,\ldots,h} : Z_i, Z_j \in \{\overline{Z}\}\ \{X \leftarrow U\} \circ \varphi(Z_i) = \{X \leftarrow U\} \circ \varphi(Z_j) \Rightarrow i = j \tag{6.135}$$

From (6.119), (6.128), (6.131), (6.129) and (6.130), and having $\{X \leftarrow U\} \circ \varphi$ mgu of (6.132) and conditions (6.134) and (6.135), by an application of (Definition), comes the intended conclusion

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash (\mathbf{def} < \overline{Z} > \overline{v : T}\ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q)\{X \leftarrow U\}\}\quad OK \tag{6.136}$$

$$((\mathbf{def} < \overline{Z} > \overline{v : T}\ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q)\{X \leftarrow U\} \equiv \mathbf{def} < \overline{Z} > \overline{v : T\{X \leftarrow U\}}\ \mathbf{in}\ M_1 \& \ldots \& M_t[P\{X \leftarrow U\}]Q\{X \leftarrow U\})$$

(**Case of Message**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash M \quad OK \tag{6.137}$$

and

$$\Gamma \vdash U :: 0 \tag{6.138}$$

(6.137) is concluded from

$$\Gamma, X :: 0, \Gamma' \vdash M : \mathbf{msg} \tag{6.139}$$

By Lemma 6.4.2, having (6.139) and (6.138), we conclude

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash M : \mathbf{msg}\{X \leftarrow U\} \tag{6.140}$$

$$(\mathbf{msg}\{X \leftarrow U\} \equiv \mathbf{msg})$$

From (6.140), by an application of (Message), we obtain

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash M \quad OK \tag{6.141}$$

Since $(M\{X \leftarrow U\} \equiv M)$, from (6.141) comes the intended conclusion

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash M\{X \leftarrow U\} \quad OK \tag{6.142}$$

(**Case of Restriction**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash \textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P \quad OK \tag{6.143}$$

and

$$\Gamma \vdash U :: 0 \tag{6.144}$$

(6.143) is concluded from

$$\Gamma, X :: 0, \Gamma', v_1 : T_1, \ldots, v_n : T_n \vdash P \quad OK \tag{6.145}$$

By induction hypothesis on (6.145) we conclude

$$\begin{array}{c} \Gamma, \Gamma'\{X \leftarrow U\}, v_1 : (T_1\{X \leftarrow U\}), \ldots, v_n : (T_n\{X \leftarrow U\}) \\ \vdash P\{X \leftarrow U\} \quad OK \end{array} \tag{6.146}$$

From (6.146), by an application of (Restriction), comes the intended conclusion

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash (\textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P)\{X \leftarrow U\} \quad OK \tag{6.147}$$

$$\begin{array}{c} ((\textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P)\{X \leftarrow U\} \\ \equiv \\ \textbf{new } v_1 : (T_1\{X \leftarrow U\}); \ldots; v_n : (T_n\{X \leftarrow U\}) \textbf{ in } P\{X \leftarrow U\}) \end{array}$$

(**Case of Parallel composition**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash P \mid Q \quad OK \tag{6.148}$$

and

$$\Gamma \vdash U :: 0 \tag{6.149}$$

(6.148) is concluded from

$$\Gamma, X :: 0, \Gamma' \vdash P \quad OK \tag{6.150}$$

and

$$\Gamma, X :: 0, \Gamma' \vdash Q \quad OK \tag{6.151}$$

By induction hypothesis on (6.150), we conclude

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash P\{X \leftarrow U\} \quad OK \tag{6.152}$$

By induction hypothesis on (6.151), we conclude

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash Q\{X \leftarrow U\} \quad OK \tag{6.153}$$

From (6.152) and (6.153), by an application of (Parallel composition), comes the intended conclusion

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash (P \mid Q)\{X \leftarrow U\} \quad OK \tag{6.154}$$

$$((P \mid Q)\{X \leftarrow U\} \equiv P\{X \leftarrow U\} \mid Q\{X \leftarrow U\})$$

(**Case of Inaction**)
We have

$$\Gamma, X :: 0, \Gamma' \vdash \mathbf{inaction} \quad OK \tag{6.155}$$

and

$$\Gamma \vdash U :: 0 \tag{6.156}$$

Since $\mathbf{inaction}\{X \leftarrow U\} \equiv \mathbf{inaction}$, the conclusion is obtained by an application of (Inaction)

$$\Gamma, \Gamma'\{X \leftarrow U\} \vdash \mathbf{inaction}\{X \leftarrow U\} \quad OK \tag{6.157}$$

6.7 Type inference

Lemma 6.7.1 *If we have $\sigma = mgu_n(S \cup V, Z)$ then exists ϕ and γ such that $\sigma = \gamma \circ \phi$, $\phi = mgu_n(S, Z')$ and $\gamma = mgu_n(\phi(V), Z'')$.*

Proof We have

$$\sigma = mgu_n(S \cup V, Z) \tag{6.158}$$

From (6.158) we conclude that σ unifies S, so S is solvable, and more precisely, S has a most general solution

$$\exists \phi . \phi = mgu_n(S, Z') \tag{6.159}$$

From (6.159), recalling Definition 6.6.2 and that σ unifies S we conclude

$$\exists \gamma . \sigma = \gamma \circ \phi \tag{6.160}$$

We must now show that γ is the most general unifier of $\phi(V)$, which is established proving that it is an unifier and that it is most general.

(**Unifier**)
As well as being an unifier of S we know that σ unifies V, so

$$\forall X = Y \in V . \sigma(X) = \sigma(Y) \tag{6.161}$$

From (6.161) and (6.160) we get that

$$\forall X = Y \in V . \gamma \circ \phi(X) = \gamma \circ \phi(Y) \tag{6.162}$$

Considering the desired set of identifications $\phi(V)$, from (6.162) we obtain

$$\forall \phi(X) = \phi(Y) \in \phi(V) . \gamma(\phi(X)) = \gamma(\phi(Y)) \tag{6.163}$$

From (6.163), through the introduction of new variables, comes the intended conclusion

$$\forall W = Z \in \phi(V) . \gamma(W) = \gamma(Z) \tag{6.164}$$

(**Most general**)
Considering Φ as an unifier of $\phi(V)$, we get that

$$\forall W = Z \in \phi(V) . \Phi(W) = \Phi(Z) \tag{6.165}$$

From (6.165), noticing that for all W, Z such that $W = Z \in \phi(V)$, there exist X, Y such that $X = Y \in V$, $W = \phi(X)$ and $Z = \phi(Y)$, we obtain

$$\forall \phi(X) = \phi(Y) \in \phi(V) . \Phi(\phi(X)) = \Phi(\phi(Y)) \tag{6.166}$$

(6.166) gives us

$$\forall X = Y \in V . \Phi \circ \phi(X) = \Phi \circ \phi(Y) \tag{6.167}$$

We can extend the identification set considered in (6.167) from V to $S \cup V$, because we know that ϕ unifies S, which leads us to

$$\forall X = Y \in S \cup V. \Phi \circ \phi(X) = \Phi \circ \phi(Y) \tag{6.168}$$

which means that $\Phi \circ \phi$ is an unifier of $S \cup V$, so, from (6.158) and Definition 6.6.2, we can conclude that

$$\exists \psi. \Phi \circ \phi = \psi \circ \sigma \tag{6.169}$$

From (6.169) and (6.160) we can write

$$\Phi \circ \phi = \psi \circ \gamma \circ \phi \tag{6.170}$$

Recalling that no other restriction was made on Φ, (6.170) leads us to the intended conclusion

$$\forall \Phi \text{ unifier of } \phi(V). \exists \psi. \Phi = \psi \circ \gamma \tag{6.171}$$

From (6.164) and (6.171) we conclude

$$\gamma = mgu_n(\phi(V), Z'') \tag{6.172}$$

The intended conclusion is therefore presented in (6.159), (6.160) and (6.172).

Lemma 6.7.2 *Having $\Phi = \{t_1, \ldots, t_w\}$, if, for all $j \in 1, \ldots, w$, the judgments $\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(t_j) \Rightarrow (S_j, U_j, Z_j)$ and $\Gamma \vdash t_j\{\overline{x \leftarrow v}\} : B_j$ are derivable, and if $\sigma = mgu_n(S_1 \cup \ldots \cup S_w, \{\overline{Y}\} \cup Z_1 \cup \ldots \cup Z_w)$ then*

- *for all i such that $x_i \in vars(\Phi)$, exists V_i such that the judgment $\Gamma \vdash v_i : V_i$ is derivable;*
- *and exists δ such that*
  - *for all $j \in 1, \ldots, w$, $B_j = \delta(\sigma(U_j))$;*
  - *for all j such that $x_j \in vars(\Phi)$, $V_j = \delta \circ \sigma(Y_j)$.*

Proof By induction on the size of the multiset (Φ).

(*Case of a single element*)

(**Case of Identifier**)

We have

$$\Phi = \{id\} \tag{6.173}$$

and

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(id) \Rightarrow (\emptyset, U, Z) \tag{6.174}$$

and

$$\Gamma \vdash id\{\overline{x \leftarrow v}\} : B \tag{6.175}$$

and

$$\sigma = mgu_n(\emptyset, \{\overline{Y}\} \cup \{\overline{Z}\}) \tag{6.176}$$

We must now consider two separate cases: either $x_i \neq id$ for all i or exists i such that $x_i = id$.

(*Case of $x_i \neq id$ for all i*)
We must have

$$B = U \tag{6.177}$$

From (6.176) we obtain

$$\sigma(U) = U \tag{6.178}$$

From (6.177) and (6.178), considering δ as the identity substitution, then it satisfies the desired property

$$B = \delta(\sigma(U)) \tag{6.179}$$

Since $vars(\Phi) \cap \{\overline{x}\} = \emptyset$, the result presented in (6.179) is the only one required for this case.

(*Case of exists i such that $x_i = id$*)
We must have

$$U = Y_i \tag{6.180}$$

From (6.175) we obtain

$$\Gamma \vdash v_i : B \tag{6.181}$$

(6.181) gives us

$$B = V_i \tag{6.182}$$

Let us consider δ such that

$$\delta(Y_i) = V_i \tag{6.183}$$

Considering (6.183) and regarding that $\sigma(Y_i) = Y_i$, we obtain

$$V_i = \delta \circ \sigma(Y_i) \tag{6.184}$$

From (6.180), (6.182) and (6.183), regarding once again that $\sigma(Y_i) = Y_i$, we conclude

$$B = \delta(\sigma(U)) \tag{6.185}$$

Since, in this case $vars(\Phi) \cap \{\overline{x}\} = x_i$, the results shown in (6.181), (6.184) and (6.185) correspond to the intended conclusion.

The intended conclusion is obtained from either (6.179) or (6.181), (6.184) and (6.185) depending if $x_i \neq id$ for all i or exists i such that $x_i = id$, respectively, being the proof shown for the former identical to the one required for the value literal case.

(**Case of Constructor**)

We have

$$\Phi = \{f(u_1, \ldots, u_n)\} \tag{6.186}$$

and

$$\begin{array}{c} \Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(f(u_1, \ldots, u_n)) \\ \Rightarrow (S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, A, \{\overline{Z}\} \cup W_1 \cup \ldots \cup W_n) \end{array} \tag{6.187}$$

and

$$\Gamma \vdash f(u_1, \ldots, u_n)\{\overline{x \leftarrow v}\} : B \tag{6.188}$$

and

$$\begin{array}{c} \sigma = mgu_n(S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, \\ \{\overline{Y}\} \cup \{\overline{Z}\} \cup W_1 \cup \ldots \cup W_n) \end{array} \tag{6.189}$$

(6.187) is concluded from

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(u_j) \Rightarrow (S_j, U_j, W_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.190}$$

and from

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash f : [\overline{Z}](T_1, \ldots, T_n)A \tag{6.191}$$

(6.188) is concluded from the existence of φ such that

$$\Gamma \vdash u_j\{\overline{x \leftarrow v}\} : \varphi(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.192}$$

and that

$$\varphi(A) = B \tag{6.193}$$

and from

$$\Gamma \vdash f : [\overline{Z}](T_1, \ldots, T_n)A \tag{6.194}$$

From (6.189), by Lemma 6.7.1, there exists ψ, γ such that

$$\sigma = \gamma \circ \psi \tag{6.195}$$

and

$$\psi = mgu_n(S_1 \cup \ldots \cup S_n, \{\overline{Y}\} \cup W_1 \cup \ldots \cup W_n) \tag{6.196}$$

and

$$\begin{array}{c} \gamma = mgu_n(\psi(\{T_1 = U_1; \ldots; T_n = U_n\}), \\ \{\overline{Y}\} \cup \{\overline{Z}\} \cup W_1 \cup \ldots \cup W_n \cup Im(\psi)) \end{array} \tag{6.197}$$

By induction hypothesis on (6.190), (6.192) and (6.196), we conclude, for all i such that $x_i \in vars(\{u_1, \ldots, u_n\})$, there exists V_i such that

$$\Gamma \vdash v_i : V_i \tag{6.198}$$

and that exists θ such that, for all $j \in 1, \ldots, n$

$$\varphi(T_j) = \theta(\psi(U_j)) \tag{6.199}$$

and that, for all i such that $x_i \in vars(\{u_1, \ldots, u_n\})$

$$V_i = \theta \circ \psi(Y_i) \tag{6.200}$$

Let us now consider substitution $\varphi + \theta$. This substitution is an unifier of $\psi(\{T_1 = U_1; \ldots; T_n = U_n\})$, if

$$(\varphi + \theta)(\psi(T_j)) = (\varphi + \theta)(\psi(U_j)) \quad \text{for all } j \in 1, \ldots, n \tag{6.201}$$

Looking at the left hand side of the equation presented in (6.201), regarding that $dom(\psi) \cap vars(T_j) = \emptyset$, for all $j \in 1, \ldots, n$, since $vars(T_j) \subseteq \{\overline{Z}\}$ and $\{\overline{Z}\} \cap vars(S_1 \cup \ldots \cup S_n) = \emptyset$, and from the latter $dom(\psi) \cap \{\overline{Z}\} = \emptyset$, we get that

$$(\varphi + \theta)(\psi(T_j)) = (\varphi + \theta)(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.202}$$

From (6.202), noticing that $dom(\theta) \cap vars(T_j) = \emptyset$, through a similar reasoning, we conclude

$$(\varphi + \theta)(\psi(T_j)) = \varphi(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.203}$$

Looking at the right hand side of the equation presented in (6.201), regarding that $\varphi(\psi(U_j)) = \psi(U_j)$, for all $j \in 1, \ldots, n$, since $dom(\varphi) = \{\overline{Z}\}$ and $\{\overline{Z}\} \cap vars(U_j) = \emptyset$, for j in the referred bounds, and $\{\overline{Z}\} \cap Im(\psi) = \emptyset$, we obtain

$$(\varphi + \theta)(\psi(U_j)) = \theta(\psi(U_j)) \quad \text{for all } j \in 1, \ldots, n \tag{6.204}$$

From (6.204) and (6.199), we conclude

$$(\varphi + \theta)(\psi(U_j)) = \varphi(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.205}$$

Combining (6.203) and (6.205) we conclude

$$(\varphi + \theta)(\psi(T_j)) = (\varphi + \theta)(\psi(U_j)) \quad \text{for all } j \in 1, \ldots, n \tag{6.206}$$

(6.206) gives us that $\varphi + \theta$ is an unifier of $\psi(\{T_1 = U_1; \ldots; T_n = U_n\})$ which allows us to conclude, from (6.197) and Definition 6.6.2, that exists δ such that

$$\varphi + \theta = \delta \circ \gamma \tag{6.207}$$

Regarding that $\{\overline{Z}\} \subseteq dom(\gamma)$, from (6.207), splitting substitution γ in regard to it's domain, we get

$$\varphi + \theta = \delta \circ (\gamma\lfloor_Z + \gamma\lfloor_{\overline{Z}}) \tag{6.208}$$

From (6.208), knowing that $dom(\varphi) = \{\overline{Z}\}$, by restricting the domains on both sides, we get

$$\theta = \delta \circ \gamma\lfloor_{\overline{Z}} \tag{6.209}$$

Recalling result (6.200) where i is such that $x_i \in vars(\{u_1, \ldots, u_n\})$, using (6.209), we conclude

$$V_i = \delta \circ \gamma\lfloor_{\overline{Z}} \circ \psi(Y_i) \tag{6.210}$$

Regarding that, for the referred bounds of i, $\gamma\lfloor_{\overline{Z}} \circ \psi(Y_i) = \gamma \circ \psi(Y_i)$, because $\{\overline{Z}\} \cap \{\overline{Y}\} = \emptyset$ and $Im(\psi) \cap \{\overline{Z}\} = \emptyset$, from (6.210) and (6.195) we conclude

$$V_i = \delta \circ \sigma(Y_i) \tag{6.211}$$

We must now show that δ is such that

$$B = \delta(\sigma(A)) \tag{6.212}$$

from (6.212), recalling that $\varphi(A) = B$ and using (6.195), we obtain

$$\varphi(A) = \delta(\gamma \circ \psi(A)) \tag{6.213}$$

Regarding that $\delta(\gamma \circ \psi(A)) \equiv (\delta \circ \gamma) \circ \psi(A)$, (6.213) is equivalent to

$$\varphi(A) = (\delta \circ \gamma) \circ \psi(A) \tag{6.214}$$

from (6.214), using (6.207), we obtain

$$\varphi(A) = (\varphi + \theta) \circ \psi(A) \tag{6.215}$$

Regarding that $\psi(A) = A$ and $\theta(A) = A$, since $vars(A) \subseteq \{\overline{Z}\}$ and $\{\overline{Z}\} \cap dom(\psi) = \emptyset$ and $\{\overline{Z}\} \cap dom(\theta) = \emptyset$, we obtain

$$\varphi(A) = \varphi(A) \tag{6.216}$$

(6.216) gives us the intended conclusion

$$B = \delta(\sigma(A)) \tag{6.217}$$

Since $vars(\Phi) = vars(\{u_1, \ldots, u_n\})$, the intended conclusion is presented in (6.198), (6.211) and (6.217).

(*Case of multiple elements*)

(**Case of Identifier as the first element**)

We have

$$\Phi = \{id, t_2, \ldots, t_w\} \tag{6.218}$$

and

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(id) \Rightarrow (\emptyset, U_1, Z) \tag{6.219}$$

and

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(t_j) \Rightarrow (S_j, U_j, Z_j) \quad \text{for all } j \in 2, \ldots, w \tag{6.220}$$

and

$$\Gamma \vdash id\{\overline{x \leftarrow v}\} : B_1 \tag{6.221}$$

and

$$\Gamma \vdash t_j\{\overline{x \leftarrow v}\} : B_j \quad \text{for all } j \in 2, \ldots, w \tag{6.222}$$

and

$$\sigma = mgu_n(S_2 \cup \ldots \cup S_w, \{\overline{Y}\} \cup Z_2 \cup \ldots \cup Z_w) \tag{6.223}$$

By induction hypothesis on (6.220), (6.222) and (6.223), we conclude that for all i such that $x_i \in vars(\{t_2, \ldots, t_w\})$, there exists V_i such that

$$\Gamma \vdash v_i : V_i \tag{6.224}$$

and that exists δ such that, for all $j \in 2, \ldots, w$

$$B_j = \delta(\sigma(U_j)) \tag{6.225}$$

and that, for all i such that $x_i \in vars(\{t_2, \ldots, t_w\})$

$$V_i = \delta \circ \sigma(Y_i) \tag{6.226}$$

We must now consider two separate cases: either $x_i \neq id$ for all i or exists i such that $x_i = id$.

(*Case of $x_i \neq id$ for all i*)
We must have

$$B_1 = U_1 \tag{6.227}$$

Regarding that $vars(U_1) \cap dom(\sigma) = \emptyset$, we obtain

$$\sigma(U_1) = U_1 \tag{6.228}$$

From (6.227) and (6.228), regarding that $\delta(B_1) = B_1$ since $vars(B_1) = \emptyset$, we conclude

$$B_1 = \delta(\sigma(U_1)) \tag{6.229}$$

Since $vars(\Phi) = vars(\{t_2, \ldots, t_w\})$, the results given by induction hypothesis and the one presented in (6.229) are the only ones required for this case.

(*Case of exists i such that $x_i = id$*)
We must have

$$U_1 = Y_i \tag{6.230}$$

From (6.221) we conclude

$$\Gamma \vdash v_i : B_1 \tag{6.231}$$

(6.231) gives us

$$B_1 = V_i \tag{6.232}$$

We must now consider two different cases: either $x_i \in vars(\{t_2, \ldots, t_w\})$ or $\{x_i\} \cap vars(\{t_2, \ldots, t_w\}) = \emptyset$.

(**Case of** $x_i \in vars(\{t_2, \ldots, t_w\})$)

In this case the results obtained through induction hypothesis give us the intended conclusion, being the coherence of V_i guaranteed by (6.224) and (6.231).

(**Case of** $\{x_i\} \cap vars(\{t_2, \ldots, t_w\}) = \emptyset$)

Let us consider θ defined by

$$\theta = \delta + \{Y_i \leftarrow V_i\} \tag{6.233}$$

Regarding that $\sigma(Y_i) = Y_i$ and that $(\delta + \{Y_i \leftarrow V_i\})Y_i = \{Y_i \leftarrow V_i\}(Y_i) = V_i$, we conclude

$$V_i = \theta \circ \sigma(Y_i) \tag{6.234}$$

From (6.226) and (6.233), regarding that $\theta \circ \sigma(Y_k) = \delta \circ \sigma(Y_k)$ when $k \neq i$, we conclude that for all j such that $x_j \in vars(\Phi)$

$$V_j = \theta \circ \sigma(Y_j) \tag{6.235}$$

Regarding that $B_1 = V_i$, $U_1 = Y_i$, $\sigma(Y_i) = Y_i$ and that $\theta(Y_i) = V_i$, we conclude

$$B_1 = \theta(\sigma(U_1)) \tag{6.236}$$

From (6.225) and (6.236), regarding that $\theta(\sigma(U_k)) = \delta(\sigma(U_k))$ when $k \neq i$, we conclude that for all $j \in 1, \ldots, w$

$$B_j = \theta(\sigma(U_j)) \tag{6.237}$$

The results required for this case are presented in (6.224), (6.231), (6.235) and (6.237).

The intended conclusion is obtained either from (6.224), (6.226) and (6.225) when exists i such that $id = x_i$ and $x_i \in vars(\{t_2, \ldots, t_w\})$, and when, along with (6.229), $x_i \neq id$ for all i (identical when considering value literals), or from (6.224), (6.231), (6.235) and (6.237) when exists i such that $id = x_i$ and $\{x_i\} \cap vars(\{t_2, \ldots, t_w\}) = \emptyset$.

(**Case of Constructor as the first element**)

We have

$$\Phi = \{f(u_1, \ldots, u_n), t_2, \ldots, t_w\} \tag{6.238}$$

and

$$\begin{gathered}\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(f(u_1, \ldots, u_n)) \\ \Rightarrow (S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\}, Q_1, \{\overline{Z}\} \cup W_1 \cup \ldots \cup W_n)\end{gathered} \tag{6.239}$$

and

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(t_j) \Rightarrow (P_j, Q_j, R_j) \quad \text{for all } j \in 2, \ldots, w \tag{6.240}$$

and

$$\Gamma \vdash f(u_1, \ldots, u_n)\{\overline{x \leftarrow v}\} : B_1 \tag{6.241}$$

and

$$\Gamma \vdash t_j\{\overline{x \leftarrow v}\} : B_j \quad \text{for all } j \in 2, \ldots, w \tag{6.242}$$

and

$$\begin{gathered}\sigma = mgu_n(S_1 \cup \ldots \cup S_n \cup \{T_1 = U_1; \ldots; T_n = U_n\} \cup P_2 \cup \ldots \cup P_w, \\ \{\overline{Y}\} \cup \{\overline{Z}\} \cup W_1 \cup \ldots \cup W_n \cup R_2 \cup \ldots \cup R_w)\end{gathered} \tag{6.243}$$

(6.239) is concluded from

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash term(u_j) \Rightarrow (S_j, U_j, W_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.244}$$

and from

$$\Gamma, \overline{Y :: 0}, \overline{x : Y} \vdash f : [\overline{Z}](T_1, \ldots, T_n)Q_1 \tag{6.245}$$

(6.241) is concluded from the existence of φ such that

$$\Gamma \vdash u_j\{\overline{x \leftarrow v}\} : \varphi(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.246}$$

and that

$$\varphi(Q_1) = B_1 \tag{6.247}$$

and from

$$\Gamma \vdash f : [\overline{Z}](T_1, \ldots, T_n)Q_1 \tag{6.248}$$

From (6.243), by Lemma 6.7.1, there exists ψ, γ such that

$$\sigma = \gamma \circ \psi \tag{6.249}$$

and

$$\psi = mgu_n(S_1 \cup \ldots \cup S_n \cup P_2 \cup \ldots \cup P_w, \{\overline{Y}\} \cup W_1 \cup \ldots \cup W_n \cup R_2 \cup \ldots \cup R_w) \tag{6.250}$$

and

$$\begin{gathered}\gamma = mgu_n(\psi(\{T_1 = U_1; \ldots; T_n = U_n\}), \\ \{\overline{Y}\} \cup \{\overline{Z}\} \cup W_1 \cup \ldots \cup W_n \cup Im(\psi))\end{gathered} \tag{6.251}$$

By induction hypothesis on (6.244), (6.240), (6.246), (6.242) and (6.250), we conclude, for all i such that $x_i \in vars(\{u_1, \ldots, u_n, t_2, \ldots, t_w\})$, there exists V_i such that

$$\Gamma \vdash v_i : V_i \tag{6.252}$$

and that exists θ such that, for all $j \in 1, \ldots, n$

$$\varphi(T_j) = \theta(\psi(U_j)) \tag{6.253}$$

and, for all $j \in 2, \ldots, w$

$$B_j = \theta(\psi(Q_j)) \tag{6.254}$$

and that, for all i such that $x_i \in vars(\{u_1, \ldots, u_n, t_2, \ldots, t_w\})$

$$V_i = \theta \circ \psi(Y_i) \tag{6.255}$$

Let us now consider substitution $\varphi + \theta$. This substitution is an unifier of $\psi(\{T_1 = U_1; \ldots; T_n = U_n\})$, if

$$(\varphi + \theta)(\psi(T_j)) = (\varphi + \theta)(\psi(U_j)) \quad \text{for all } j \in 1, \ldots, n \tag{6.256}$$

Looking at the left hand side of the equation presented in (6.256), regarding that $dom(\psi) \cap vars(T_j) = \emptyset$, for all $j \in 1, \ldots, n$, since $vars(T_j) \subseteq \{\overline{Z}\}$ and $\{\overline{Z}\} \cap vars(S_1 \cup \ldots \cup S_n \cup P_2 \cup \ldots P_w) = \emptyset$, and from the latter $dom(\psi) \cap \{\overline{Z}\} = \emptyset$, we get that

$$(\varphi + \theta)(\psi(T_j)) = (\varphi + \theta)(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.257}$$

From (6.257), noticing that $dom(\theta) \cap vars(T_j) = \emptyset$, through a similar reasoning, we conclude

$$(\varphi + \theta)(\psi(T_j)) = \varphi(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.258}$$

Looking at the right hand side of the equation presented in (6.256), regarding that $\varphi(\psi(U_j)) = \psi(U_j)$, for all $j \in 1, \ldots, n$, since $dom(\varphi) = \{\overline{Z}\}$ and $\{\overline{Z}\} \cap vars(U_j) = \emptyset$, for j in the referred bounds, and $\{\overline{Z}\} \cap Im(\psi) = \emptyset$, we obtain

$$(\varphi + \theta)(\psi(U_j)) = \theta(\psi(U_j)) \quad \text{for all } j \in 1, \ldots, n \tag{6.259}$$

From (6.259) and (6.253), we conclude

$$(\varphi + \theta)(\psi(U_j)) = \varphi(T_j) \quad \text{for all } j \in 1, \ldots, n \tag{6.260}$$

Combining (6.258) and (6.260) we conclude

$$(\varphi + \theta)(\psi(T_j)) = (\varphi + \theta)(\psi(U_j)) \quad \text{for all } j \in 1, \ldots, n \tag{6.261}$$

(6.261) gives us that $\varphi + \theta$ is an unifier of $\psi(\{T_1 = U_1; \ldots; T_n = U_n\})$ which allows us to conclude, from (6.251) and Definition 6.6.2, that exists δ such that

$$\varphi + \theta = \delta \circ \gamma \tag{6.262}$$

Regarding that $\{\overline{Z}\} \subseteq dom(\gamma)$, from (6.262), splitting substitution γ in regard to it's domain, we get

$$\varphi + \theta = \delta \circ (\gamma\lfloor_Z + \gamma\lfloor_{\overline{Z}}) \tag{6.263}$$

From (6.263), knowing that $dom(\varphi) = \{\overline{Z}\}$, by restricting the domains on both sides, we get

$$\theta = \delta \circ \gamma\lfloor_{\overline{Z}} \tag{6.264}$$

Recalling result (6.255) where i is such that $x_i \in vars(\{u_1, \ldots, u_n, t_2, \ldots, t_w\})$, using (6.264), we conclude

$$V_i = \delta \circ \gamma\lfloor_{\overline{Z}} \circ \psi(Y_i) \tag{6.265}$$

Regarding that, for the referred bounds of i, $\gamma\lfloor_{\overline{Z}} \circ \psi(Y_i) = \gamma \circ \psi(Y_i)$, because $\{\overline{Z}\} \cap \{\overline{Y}\} = \emptyset$ and $Im(\psi) \cap \{\overline{Z}\} = \emptyset$, from (6.265) and (6.249) we conclude

$$V_i = \delta \circ \sigma(Y_i) \tag{6.266}$$

We must now show that δ is such that, for all $j \in 1, \ldots, w$

$$B_j = \delta(\sigma(Q_j)) \quad \text{for all } j \in 2, \ldots, w \tag{6.267}$$

Let us consider two separate cases: either $j = 1$ or $j \in 2, \ldots, w$.

(*Case of* $j = 1$)

We intend to prove

$$B_1 = \delta(\sigma(Q_1)) \tag{6.268}$$

From (6.268), recalling that $\varphi(Q_1) = B_1$ and using (6.249), we obtain

$$\varphi(Q_1) = \delta(\gamma \circ \psi(Q_1)) \tag{6.269}$$

Regarding that $\delta(\gamma \circ \psi(Q_1)) \equiv (\delta \circ \gamma) \circ \psi(Q_1)$, (6.269) is equivalent to

$$\varphi(Q_1) = (\delta \circ \gamma) \circ \psi(Q_1) \tag{6.270}$$

from (6.270), using (6.262), we obtain

$$\varphi(Q_1) = (\varphi + \theta) \circ \psi(Q_1) \tag{6.271}$$

Regarding that $\psi(Q_1) = Q_1$ and $\theta(Q_1) = Q_1$, since $vars(Q_1) \subseteq \{\overline{Z}\}$ and $\{\overline{Z}\} \cap dom(\psi) = \emptyset$ and $\{\overline{Z}\} \cap dom(\theta) = \emptyset$, we obtain

$$\varphi(Q_1) = \varphi(Q_1) \tag{6.272}$$

from (6.272) we conclude

$$B_1 = \delta(\sigma(Q_1)) \tag{6.273}$$

(*Case of* $j \in 2, \ldots, w$)

We intend to prove

$$B_j = \delta(\sigma(Q_j)) \tag{6.274}$$

From (6.274), using (6.249), we obtain

$$B_j = \delta(\gamma \circ \psi(Q_j)) \tag{6.275}$$

Since $\delta(\gamma \circ \psi(Q_j)) \equiv (\delta \circ \gamma) \circ \psi(Q_j)$, (6.275) is equivalent to

$$B_j = (\delta \circ \gamma) \circ \psi(Q_j) \tag{6.276}$$

From (6.276), using (6.262), we obtain

$$B_j = (\varphi + \theta) \circ \psi(Q_j) \tag{6.277}$$

Regarding that $\varphi(\psi(Q_j)) = \psi(Q_j)$, from (6.277) we obtain

$$B_j = \theta \circ \psi(Q_j) \tag{6.278}$$

Since (6.254) proves (6.278) we conclude

$$B_j = \delta(\sigma(Q_j)) \tag{6.279}$$

Joining (6.273) and (6.279) we obtain

$$B_j = \delta(\sigma(Q_j)) \quad \text{for all } j \in 1, \ldots, w \tag{6.280}$$

Since $vars(\Phi) = vars(\{u_1, \ldots, u_n, t_2, \ldots, t_w\})$, the intended conclusion is presented in (6.252), (6.266) and (6.280).

6.8 Subject Reduction

Lemma 6.8.1 *If the judgment* $\Gamma \vdash P \quad OK$ *is derivable and* $\Gamma \vdash P \rightarrow \Gamma \vdash Q$*, conformingly to the language's operational semantics, then the judgment* $\Gamma \vdash Q \quad OK$ *is also derivable.*

Proof By induction on the structure of the reduction derivation.

(**Case of Restriction reduction**)
We have

$$\Gamma \vdash \textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P \quad OK \tag{6.281}$$

and

$$\begin{array}{c} \Gamma \vdash \textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } P \quad OK \\ \rightarrow \Gamma \vdash \textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } Q \quad OK \end{array} \tag{6.282}$$

(6.281) is concluded from

$$\Gamma, v_1 : T_1, \ldots, v_n : T_n \vdash P \tag{6.283}$$

and (6.282) is concluded from

$$\Gamma, v_1 : T_1, \ldots, v_n : T_n \vdash P \rightarrow \Gamma, v_1 : T_1, \ldots, v_n : T_n \vdash Q \tag{6.284}$$

By induction hypothesis on (6.283) and (6.284), we conclude

$$\Gamma, v_1 : T_1, \ldots, v_n : T_n \vdash Q \tag{6.285}$$

From (6.285), by an application of (Restriction), comes the intended conclusion

$$\Gamma \vdash \textbf{new } v_1 : T_1; \ldots; v_n : T_n \textbf{ in } Q \quad OK \tag{6.286}$$

(**Case of Composition reduction**)
We have

$$\Gamma \vdash P \mid R \quad OK \tag{6.287}$$

and

$$\Gamma \vdash P \mid R \rightarrow \Gamma \vdash Q \mid R \tag{6.288}$$

(6.287) is concluded from

$$\Gamma \vdash P \quad OK \tag{6.289}$$

and

$$\Gamma \vdash R \quad OK \tag{6.290}$$

and (6.288) is concluded from

$$\Gamma \vdash P \rightarrow \Gamma \vdash Q \tag{6.291}$$

By induction hypothesis on (6.289) and (6.291), we conclude

$$\Gamma \vdash Q \quad OK \tag{6.292}$$

From (6.292) and (6.290), by an application of (Parallel composition), comes the intended conclusion

$$\Gamma \vdash Q \mid R \quad OK \tag{6.293}$$

(**Case of Definition reduction**)
We have

$$\begin{array}{c}\Gamma \vdash m_1 \mid \cdots \mid m_t \mid \mathbf{def} < X_1, \ldots, X_n > v_1 : T_1; \ldots; v_k : T_k \\ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q \quad OK\end{array} \tag{6.294}$$

and

$$\begin{array}{c}\Gamma \vdash m_1 \mid \cdots \mid m_t \mid \mathbf{def} < X_1, \ldots, X_n > v_1 : T_1; \ldots; v_k : T_k \\ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q \quad OK \\ \rightarrow \\ \Gamma \vdash \sigma(Q) \\ \mid \mathbf{def} < X_1, \ldots, X_n > v_1 : T_1; \ldots; v_k : T_k\ \mathbf{in}\ M_1 \& \ldots \& M_t[P]Q \quad OK\end{array} \tag{6.295}$$

(6.295) is concluded from

$$\sigma(M_i) = m_i \text{ (for } i = 1, \ldots, t) \tag{6.296}$$

and

$$\Gamma \vdash \sigma(P) \xrightarrow{*} \surd \tag{6.297}$$

where $\sigma : X \rightarrow \mathtt{T}, v \rightarrow \mathcal{T}$, meaning that σ comprehends in it's domain type variables (X) and value variables (v) which are mapped to types ($\mathtt{T}$) and values ($\mathcal{T}$), respectively.

From (6.294), regarding the derivation of the definition, we obtain

$$replace(\{M_1, \ldots, M_t\}, \{v_1, \ldots, v_k\}) \Rightarrow (\{M'_1, \ldots, M'_t\}, \{v'_1, \ldots, v'_m\}, \gamma) \tag{6.298}$$

and

$$\begin{array}{c}\Gamma, Y_1 :: 0, \ldots, Y_m :: 0, v'_1 : Y_1, \ldots, v'_m : Y_m \vdash term(M'_j) \Rightarrow (S_j, \mathbf{msg}, Z_j) \\ \text{for all } j \in 1, \ldots, t\end{array} \tag{6.299}$$

and

$$\theta = mgu_n(S_1 \cup \ldots \cup S_t, \{\overline{Y}\} \cup Z_1 \cup \ldots \cup Z_t) \tag{6.300}$$

and

$$\varphi = mgu_n(\bigcup_{q:Y_q \in dom(\theta) \wedge \exists_l . \gamma(v'_q) = v_l} \{T_l = \theta(Y_q)\}, \{\overline{X}\}) \tag{6.301}$$

and

$$\Gamma, X_1 :: 0, \ldots, X_n :: 0, v_1 : T_1, \ldots, v_k : T_k \vdash P \quad OK \tag{6.302}$$

and

$$\Gamma, X_1 :: 0, \ldots, X_n :: 0, v_1 : T_1, \ldots, v_k : T_k \vdash Q \quad OK \tag{6.303}$$

Regarding the domain of σ it is possible to establish from it a sum of substitutions, respective to either types ($\sigma\lfloor_t$) or to values ($\sigma\lfloor_v$)

$$\sigma = \sigma\lfloor_t + \sigma\lfloor_v \tag{6.304}$$

From (6.296) and (6.304), regarding that value substitution is the only one to take place in (6.296), we conclude

$$\sigma\lfloor_v(M_i) = m_i \quad \text{for all } i \in 1, \ldots, t \tag{6.305}$$

(6.294) gives us, from the derivation of the messages, that

$$\Gamma \vdash m_i \quad OK \quad \text{for all } i \in 1, \ldots, t \tag{6.306}$$

From (6.305) and (6.306) we obtain

$$\Gamma \vdash \sigma\lfloor_v(M_i) \quad OK \quad \text{for all } i \in 1, \ldots, t \tag{6.307}$$

Knowing that $\sigma\lfloor_v$ is of the form $\{v_1 \leftarrow c_1, \ldots, v_k \leftarrow c_k\}$, from the derivation of (6.307) we obtain

$$\Gamma \vdash M_i\{v_1 \leftarrow c_1, \ldots, v_k \leftarrow c_k\} : \mathbf{msg} \quad \text{for all } i \in 1, \ldots, t \tag{6.308}$$

Regarding that for all $i \in 1, \ldots, t$ $M_i = M'_i\{v'_1 \leftarrow \gamma(v'_1), \ldots, v'_m \leftarrow \gamma(v'_m)\}$, from (6.308) we conclude

$$\begin{array}{c} \Gamma \vdash (M'_i\{v'_1 \leftarrow \gamma(v'_1), \ldots, v'_m \leftarrow \gamma(v'_m)\}) \\ \{v_1 \leftarrow c_1, \ldots, v_k \leftarrow c_k\} : \mathbf{msg} \quad \text{for all } i \in 1, \ldots, t \end{array} \tag{6.309}$$

(6.309) can be written as

$$\Gamma \vdash M'_i\{v'_1 \leftarrow \sigma\lfloor_v(\gamma(v'_1)), \ldots, v'_m \leftarrow \sigma\lfloor_v(\gamma(v'_m))\} : \mathbf{msg} \quad \text{for all } i \in 1, \ldots, t \tag{6.310}$$

By Lemma 6.7.2 considering that $\Phi = \{M'_1, \ldots, M'_t\}$, from (6.299), (6.310) and (6.300) we conclude that for all i such that $v'_i \in vars(\Phi)$ and $\gamma(v'_i) = v_j$, exists A_j such that

$$\Gamma \vdash c_j : A_j \tag{6.311}$$

and that exists δ, such that, for all i such that $v'_i \in vars(\Phi)$ and $\gamma(v'_i) = v_j$

$$A_j = \delta \circ \theta(Y_i) \tag{6.312}$$

From (6.301) we have that

$$\varphi(T_j) = \varphi(\theta(Y_i)) \quad \text{for } j, i \text{ such that } Y_i \in dom(\theta) \wedge \gamma(v_i') = v_j \tag{6.313}$$

from (6.313), knowing that $dom(\varphi) = \{\overline{X}\}$, we obtain

$$\varphi(T_j) = \theta(Y_i) \quad \text{for } j, i \text{ such that } Y_i \in dom(\theta) \wedge \gamma(v_i') = v_j \tag{6.314}$$

Regarding that for all i such that $v_i' \in vars(\Phi)$ we have that $Y_i \in dom(\theta)$ and that exists j such that $\gamma(v_i') = v_j$, from (6.312) and (6.314) we obtain

$$A_j = \delta \circ \varphi(T_j) \tag{6.315}$$

From (6.315) and (6.311) we conclude

$$\Gamma \vdash c_j : \delta \circ \varphi(T_j) \tag{6.316}$$

From (6.316) we obtain $\sigma\lfloor_t$, which is to say

$$\sigma\lfloor_t = \delta \circ \varphi \tag{6.317}$$

From (6.303), by Lemma 6.6.4 (n times), regarding that $\Gamma \vdash \sigma\lfloor_t(X_i) :: 0$, for $i \in 1, \ldots, n$, we conclude

$$\Gamma, v_1 : \sigma\lfloor_t(T_1), \ldots, v_k : \sigma\lfloor_t(T_k) \vdash \sigma\lfloor_t(Q) \quad OK \tag{6.318}$$

From (6.318), by Lemma 6.3.1 (k times), regarding (6.316), we conclude

$$\Gamma \vdash \sigma\lfloor_v(\sigma\lfloor_t(Q)) \quad OK \tag{6.319}$$

From (6.319) and (6.304) we obtain

$$\Gamma \vdash \sigma(Q) \quad OK \tag{6.320}$$

From the derivation of (6.294) we have

$$\Gamma \vdash \textbf{def} < X_1, \ldots, X_n > v_1 : T_1; \ldots; v_k : T_k \textbf{in } M_1 \& \ldots \& M_t[P]Q \quad OK \tag{6.321}$$

From (6.320) and (6.321), by an application of (Parallel composition), comes the intended conclusion

$$\Gamma \vdash \sigma(Q) \mid \textbf{def} < X_1, \ldots, X_n > v_1 : T_1; \ldots; v_k : T_k \textbf{in } M_1 \& \ldots \& M_t[P]Q \quad OK \tag{6.322}$$

Chapter 7

Closing remarks

The development of the POLY static checker contributed mainly to the study of types in concurrent based languages. A main focus of the presented work was the introduction of polymorphism in the language, which was a task that revealed to be somewhat tricky.

The more time consuming study went to the acceptance pattern verification. The first path that was pursued was, as a first approach usually is, mostly brute force driven, being the idea then, to consider generic types associated with each type parameter encountered in the pattern, being unnecessary generality assured.

After that the ideas came close to their final form, except when it came to an important assertion, that we believed was true at that point, which was considering that the same variable usage in two pattern terms would guarantee that they could share the same type variables, because the same value would be present in the matching candidate messages. This assertion was denied when we came to realize that syntactic equality does not assure type equality, meaning that the same identifier may occur associated to different types, because of the existence of polymorphic values in POLY, such as **nil**.

Finally we arrived at the presented considerations, which we believe to be an elegant form of handling universal polymorphism in this language.

The POLY language development was, therefore, useful in the better understanding of types in pattern matching based languages in a concurrent setting, starting at the more simple issues involved in such contexts but specially in making some sense of the usage of polymorphism in such languages, always bearing in mind the guarantees that types offer in assuring error free programs.

Bibliography

[App98] Andrew Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[Cai99] Luís Caires. *A Model for Declarative Programming and Specification with Concurrency and Mobility*. PhD thesis, Dept. de Informática, FCT, Universidade Nova de Lisboa, 1999.

[CUP] http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[Jav] http://java.sun.com/.

[Lex] http://www.cs.princeton.edu/ appel/modern/java/JLex/.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH91] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standardt ML*. MIT Press, 1991.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

Appendix A

User manual

In order to give some helpful information about how to use the developed POLY static checker, this appendix outlines the main procedures involved in the usage of this application.

Firstly the required software, which consists in an installation of a java development kit [Jav], and an installation of the referred parsing tools JLex [Lex] and CUP [CUP]. Refer to the installation instructions of these distributions for help on this procedure.

The static checker is available in two versions, one dedicated to windows and other to linux. They only differ in the presented <README> file, in some helpful files and in the application used to compress the provided files. The different files consist no more than useful linux shell scripts and DOS batch files, used in generating the static checker's parser (<genSource>) and in compiling the hole application (<compile>). All these instructions are presented in detail in the <README> files, with a special remark for the windows version regarding the specification of the <CLASSPATH> environment variable.

After decompressing the <PolyC> package a directory tree is established, being present in the root directory <POLY> the <README> and the scripts, in the <bin> directory the Java code, in the <source> directory the source code and in the <doc> directory the Javadoc generated documentation.

Finally we are able to use the static checker. Of course POLY source code must be developed to provide input to the application, unless your intent is simply to try it out, and in that case sample POLY programs that are present in the <bin> directory can be used, contained in files with extensions <.poly>. The referred extension is required for the files that we wish to provide the static checker as input. A usage example is shown in (A.1).

```
java PolyC.Main program.poly                    (A.1)
```

After running the application one of two things will occur: the program provided as input is syntactically and typing correct or it is not. In the first

```
******************

Parsing successful

******************

******************

Type verification successful

******************
```

Figure A.1: Successful parsing and type checking presentation.

```
***
Syntax error: Found 'def' in line 9, column 0
* Expecting *
|
* Context *
def <X> x:X in conx(x, nil) [] inaction
def <Y> x, y:Y; l:list[Y] in conx(x, cons(y, l)) [] inaction
|
***
```

Figure A.2: Syntactic error presentation.

case the messages presented in Figure A.1 will appear and in the second or a syntax error (Figure A.2) is shown or at least one type error is presented (Figure A.3).

There are two optional arguments for the application which are `<-d>` after the filename specification, that produces debug information, and `<-h>` before the filename, that gives application usage information. The debug information that is presented, when the `<-d>` argument is provided to the application, regards the referred equation systems that the type checker must solve in order to ensure type validity, both in the verification of messages

```
******************

Parsing successful

******************

***
Found Type Error while evaluating agents pattern present in line 26, column 29
* Variable declarations and usage makes pattern typing incoherent *
* Context *

def <X> x, y:X; l:list[X] in op(x, l, y, l) [] inaction
|
***
```

Figure A.3: Type error presentation.

```
******************

***Verifying Definition***

Line: 8 Column: 0

***



***Term Equations***

Line: 8 Column: 15


Equation 0
ASTBaseType: TypeId: list [ASTBaseType: TypeId: $Param1 ]
ASTBaseType: TypeId: list [ASTBaseType: TypeId: $Param3 ]

Equation 1
ASTBaseType: TypeId: $Param1
ASTBaseType: TypeId: $Param2
***



***Pattern Equations***


Equation 0
ASTBaseType: TypeId: X
ASTBaseType: TypeId: $open$Param4
***
```

Figure A.4: Type error presentation.

and in the verification of acceptance patterns.

The illustration presented in Figure A.4 shows an example extracted from a debug output, where are present the equation systems, containing type form identifications, where type parameter identifiers may surface, easily identified in the example, because they all contain `$Param` in their string.

After that come the usual programming tasks, like debugging, fixing and running...

Appendix B

Grammar

```
===== Terminals =====
[0]EOF [1]error [2]BINOP0 [3]BINOP1 [4]BINOP2
[5]BINOP3 [6]BINOP4 [7]BINOP5 [8]BINOP6 [9]BINOP7
[10]BINOP8 [11]BINOP9 [12]PREOP0 [13]PREOP1 [14]PREOP2
[15]PREOP3 [16]PREOP4 [17]PREOP5 [18]PREOP6 [19]PREOP7
[20]PREOP8 [21]PREOP9 [22]POSTOP0 [23]POSTOP1 [24]POSTOP2
[25]POSTOP3 [26]POSTOP4 [27]POSTOP5 [28]POSTOP6 [29]POSTOP7
[30]POSTOP8 [31]POSTOP9 [32]COMMA [33]NAME [34]INTEGER
[35]REAL [36]STRING [37]NIL [38]TRUE [39]FALSE
[40]PARL [41]PARR [42]SQUAREBRL [43]SQUAREBRR [44]COLON
[45]LT [46]GT [47]SEMI [48]AMPERSAND [49]COM
[50]IN [51]DEF [52]NEW [53]PIPE [54]INACTION
[55]PREFIX [56]INFIX [57]SUFFIX [58]OPERATOR [59]DECLARE
[60]TYPE [61]INTTYPE [62]BOOLTYPE [63]REALTYPE [64]STRINGTYPE
[65]MSGTYPE [66]LISTTYPE

=====Non terminals =====
[0]$START [1]terms [2]types [3]type_optional [4]arg_optional
[5]type_args [6]type_pars [7]id_type [8]primitive_type [9]typeL
[10]base_type [11]constructor_type [12]type [13]long_type [14]
    integerL
[15]realL [16]stringL [17]booleanL [18]listL [19]simple_term
[20]term [21]value_literal [22]type_arg [23]test_body [24]test
[25]ids [26]vars [27]decls [28]decls_op [29]message
[30]input [31]proc_zone [32]basic_command [33]command [34]
    definition
[35]restriction [36]simple [37]process [38]type_decl [39]
    symbol_decl
[40]declaration [41]declarations [42]unit [43]binop [44]preop
[45]postop [46]colon_op [47]assoc_spec [48]prec_spec [49]
    identifier
[50]optional [51]NT$0 [52]NT$1 [53]NT$2

===== Productions =====
[0] $START ::= unit EOF
[1] integerL ::= INTEGER
[2] realL ::= REAL
```

```
[3] stringL ::= STRING
[4] booleanL ::= TRUE
[5] booleanL ::= FALSE
[6] listL ::= NIL
[7] value_literal ::= integerL
[8] value_literal ::= realL
[9] value_literal ::= stringL
[10] value_literal ::= booleanL
[11] value_literal ::= listL
[12] typeL ::= INTTYPE
[13] typeL ::= REALTYPE
[14] typeL ::= STRINGTYPE
[15] typeL ::= MSGTYPE
[16] typeL ::= LISTTYPE
[17] typeL ::= BOOLTYPE
[18] simple_term ::= NAME
[19] simple_term ::= value_literal
[20] simple_term ::= NAME PARL terms PARR
[21] simple_term ::= binop PARL term COMMA term PARR
[22] simple_term ::= postop PARL term PARR
[23] simple_term ::= PARL term PARR
[24] binop ::= BINOP0
[25] binop ::= BINOP1
[26] binop ::= BINOP2
[27] binop ::= BINOP3
[28] binop ::= BINOP4
[29] binop ::= BINOP5
[30] binop ::= BINOP6
[31] binop ::= BINOP7
[32] binop ::= BINOP8
[33] binop ::= BINOP9
[34] preop ::= PREOP0
[35] preop ::= PREOP1
[36] preop ::= PREOP2
[37] preop ::= PREOP3
[38] preop ::= PREOP4
[39] preop ::= PREOP5
[40] preop ::= PREOP6
[41] preop ::= PREOP7
[42] preop ::= PREOP8
[43] preop ::= PREOP9
[44] postop ::= POSTOP0
[45] postop ::= POSTOP1
[46] postop ::= POSTOP2
[47] postop ::= POSTOP3
[48] postop ::= POSTOP4
[49] postop ::= POSTOP5
[50] postop ::= POSTOP6
[51] postop ::= POSTOP7
[52] postop ::= POSTOP8
[53] postop ::= POSTOP9
[54] term ::= simple_term
[55] term ::= simple_term binop term
[56] term ::= preop term
```

```
[57] term ::= term postop
[58] terms ::= term
[59] terms ::= terms COMMA term
[60] type_optional ::=
[61] type_optional ::= SQUAREBRL types SQUAREBRR
[62] primitive_type ::= typeL
[63] id_type ::= NAME
[64] id_type ::= primitive_type
[65] base_type ::= id_type type_optional
[66] types ::= type
[67] types ::= types COMMA type
[68] constructor_type ::= PARL types PARR base_type
[69] type ::= base_type
[70] type ::= constructor_type
[71] type_pars ::= LT type_args GT
[72] test_body ::=
[73] test_body ::= process
[74] test ::= SQUAREBRL test_body SQUAREBRR
[75] ids ::= NAME
[76] ids ::= ids COMMA NAME
[77] vars ::= ids COLON type
[78] decls ::= vars
[79] decls ::= decls SEMI vars
[80] decls_op ::=
[81] decls_op ::= decls
[82] message ::= term
[83] input ::=
[84] input ::= message
[85] input ::= input AMPERSAND message
[86] basic_command ::= input test simple
[87] command ::= COM basic_command
[88] command ::= COM decls IN basic_command
[89] command ::= COM type_pars decls_op IN basic_command
[90] definition ::= DEF basic_command
[91] definition ::= DEF decls IN basic_command
[92] definition ::= DEF type_pars decls_op IN basic_command
[93] restriction ::= NEW decls IN simple
[94] simple ::= INACTION
[95] simple ::= command
[96] simple ::= definition
[97] simple ::= restriction
[98] simple ::= message
[99] simple ::= PARL process PARR
[100] process ::= simple
[101] process ::= process PIPE simple
[102] assoc_spec ::= PREFIX
[103] assoc_spec ::= INFIX
[104] assoc_spec ::= SUFFIX
[105] prec_spec ::= INTEGER
[106] identifier ::= NAME
[107] identifier ::= OPERATOR
[108] optional ::=
[109] optional ::= assoc_spec prec_spec
[110] type_arg ::= NAME
```

```
[111] type_args ::= type_arg
[112] type_args ::= type_args COMMA type_arg
[113] arg_optional ::=
[114] arg_optional ::= SQUAREBRL type_args SQUAREBRR
[115] long_type ::=
[116] long_type ::= BINOP1 type
[117] NT$0 ::=
[118] NT$1 ::=
[119] type_decl ::= NT$0 TYPE NAME arg_optional long_type NT$1
    SEMI
[120] colon_op ::=
[121] colon_op ::= COLON
[122] NT$2 ::=
[123] symbol_decl ::= DECLARE identifier colon_op arg_optional
    type optional NT$2 SEMI
[124] declaration ::= type_decl
[125] declaration ::= symbol_decl
[126] declarations ::=
[127] declarations ::= declarations declaration
[128] proc_zone ::=
[129] proc_zone ::= process
[130] unit ::= declarations proc_zone
-------- Generated by CUP v0.10k Parser --------
```

Appendix C

Class listing

Hierarchy For All Packages

```
Package Hierarchies:
   PolyC, PolyC.AST, PolyC.AST.Constructs, PolyC.AST.
       Constructs.Terms,
   PolyC.AST.Types, PolyC.Environment, PolyC.Exceptions, PolyC
       .Parser,
   PolyC.Visitor
```

```
Class Hierarchy

  o class java.lang.Object
      o class PolyC.AST.ArgList
      o class PolyC.AST.Types.ASTBaseType (implements PolyC.
        AST.Types.IASTType)
      o class PolyC.AST.Constructs.ASTBasicCommand (implements
         PolyC.AST.IASTNode)
      o class PolyC.AST.Constructs.ASTCommand (implements
        PolyC.AST.IASTNode)
      o class PolyC.AST.Constructs.Terms.ASTCompound (
        implements PolyC.AST.Constructs.Terms.IASTTerm)
      o class PolyC.AST.Types.ASTCompType (implements PolyC.
        AST.Types.IASTType)
      o class PolyC.AST.Constructs.ASTDecs (implements PolyC.
        AST.IASTNode)
      o class PolyC.AST.Constructs.ASTDefinition (implements
        PolyC.AST.IASTNode)
      o class PolyC.AST.Constructs.Terms.ASTFalse (implements
        PolyC.AST.Constructs.Terms.IASTTerm)
      o class PolyC.AST.Constructs.ASTId (implements PolyC.AST
        .IASTNode)
      o class PolyC.AST.Constructs.ASTIds (implements PolyC.
        AST.IASTNode)
```

```
o class PolyC.AST.Constructs.ASTInaction (implements
    PolyC.AST.IASTNode)
o class PolyC.AST.Constructs.Terms.ASTInteger (
    implements PolyC.AST.Constructs.Terms.IASTTerm)
o class PolyC.AST.Constructs.ASTMessages (implements
    PolyC.AST.IASTNode)
o class PolyC.AST.Constructs.ASTModule (implements PolyC
    .AST.IASTNode)
o class PolyC.AST.Constructs.Terms.ASTNil (implements
    PolyC.AST.Constructs.Terms.IASTTerm)
o class PolyC.AST.Constructs.ASTOpDeclaration (
    implements PolyC.AST.IASTNode)
o class PolyC.AST.Constructs.ASTParComp (implements
    PolyC.AST.IASTNode)
o class PolyC.AST.Constructs.Terms.ASTReal (implements
    PolyC.AST.Constructs.Terms.IASTTerm)
o class PolyC.AST.Constructs.ASTRestriction (implements
    PolyC.AST.IASTNode)
o class PolyC.AST.Constructs.Terms.ASTString (implements
     PolyC.AST.Constructs.Terms.IASTTerm)
o class PolyC.AST.Constructs.Terms.ASTTrue (implements
    PolyC.AST.Constructs.Terms.IASTTerm)
o class PolyC.AST.Constructs.ASTTypeDeclaration (
    implements PolyC.AST.IASTNode)
o class PolyC.AST.Types.ASTTypeId (implements PolyC.AST.
    Types.IASTType)
o class PolyC.AST.Constructs.ASTVarDec (implements PolyC
    .AST.IASTNode)
o class PolyC.AST.Constructs.ASTVars (implements PolyC.
    AST.IASTNode)
o class PolyC.Parser.Declaration
o class PolyC.Environment.ElemList
o class PolyC.Visitor.EqualCell
o class PolyC.Visitor.EqualList
o class PolyC.Exceptions.ErrorCode
o class PolyC.Environment.HashList
o class java_cup.runtime.lr_parser
     o class PolyC.Parser.Parser
o class PolyC.Main
o class PolyC.Visitor.MGU
o class PolyC.Visitor.NameGen
o class PolyC.Parser.Symbols
o class java.lang.Throwable (implements java.io.
    Serializable)
     o class java.lang.Exception
          o class PolyC.Exceptions.IVException
               o class PolyC.Exceptions.TypeException
               o class PolyC.Exceptions.
                   UndeclaredException
          o class PolyC.Exceptions.NonUnifiableException
o class PolyC.AST.TokenValue
o class PolyC.Environment.Type (implements PolyC.
    Environment.IValue, PolyC.Visitor.IVReturn)
o class PolyC.Visitor.TypeCheck (implements PolyC.
```

```
            Visitor.IVisitor)
        o class PolyC.Environment.TypeEnvironment (implements
            PolyC.Environment.IEnvironment)
        o class PolyC.AST.TypeList
        o class PolyC.Environment.ValueEnvironment (implements
            PolyC.Environment.IEnvironment)

Interface Hierarchy

  o interface PolyC.AST.IASTNode
      o interface PolyC.AST.Constructs.Terms.IASTTerm
      o interface PolyC.AST.Types.IASTType
  o interface PolyC.Environment.IEnvironment
  o interface PolyC.Environment.IValue
  o interface PolyC.Visitor.IVisitor
  o interface PolyC.Visitor.IVReturn

------------------ Generated by javadoc ------------------
```

Appendix D

IVisitor.java

```
package PolyC.Visitor;

import PolyC.AST.*;
import PolyC.AST.Types.*;
import PolyC.AST.Constructs.*;
import PolyC.AST.Constructs.Terms.*;
import PolyC.Exceptions.*;

/**
 * Interface that represents the base of visitors of the
     abstract syntactic tree objects
 */

public interface IVisitor {

    /**
     * Visit for ASTBaseType
     */
    public IVReturn visit(ASTBaseType ast) throws IVException;

    /**
     * Visit for ASTBasicCommand
     */
    public IVReturn visit(ASTBasicCommand ast) throws
        IVException;

    /**
     * Visit for ASTCommand
     */
    public IVReturn visit(ASTCommand ast) throws IVException;

    /**
     * Visit for ASTCompType
     */
    public IVReturn visit(ASTCompType ast) throws IVException;

    /**
```

```
     * Visit for ASTCompound
     */
    public IVReturn visit(ASTCompound ast) throws IVException;

    /**
     * Visit for ASTDecs
     */
    public IVReturn visit(ASTDecs ast) throws IVException;

    /**
     * Visit for ASTDefinition
     */
    public IVReturn visit(ASTDefinition ast) throws IVException;

    /**
     * Visit for ASTFalse
     */
    public IVReturn visit(ASTFalse ast) throws IVException;

    /**
     * Visit for ASTId
     */
    public IVReturn visit(ASTId ast) throws IVException;

    /**
     * Visit for ASTIds
     */
    public IVReturn visit(ASTIds ast) throws IVException;

    /**
     * Visit for ASTInaction
     */
    public IVReturn visit(ASTInaction ast) throws IVException;

    /**
     * Visit for ASTInteger
     */
    public IVReturn visit(ASTInteger ast) throws IVException;

    /**
     * Visit for ASTMessages
     */
    public IVReturn visit(ASTMessages ast) throws IVException;

    /**
     * Visit for ASTModule
     */
    public IVReturn visit(ASTModule ast) throws IVException;

    /**
     * Visit for ASTNil
     */
    public IVReturn visit(ASTNil ast) throws IVException;
```

```
    /**
     * Visit for ASTOpDeclaration
     */
    public IVReturn visit(ASTOpDeclaration ast) throws
        IVException;

    /**
     * Visit for ASTParComp
     */
    public IVReturn visit(ASTParComp ast) throws IVException;

    /**
     * Visit for ASTReal
     */
    public IVReturn visit(ASTReal ast) throws IVException;

    /**
     * Visit for ASTRestriction
     */
    public IVReturn visit(ASTRestriction ast) throws IVException
        ;

    /**
     * Visit for ASTString
     */
    public IVReturn visit(ASTString ast) throws IVException;

    /**
     * Visit for ASTTrue
     */
    public IVReturn visit(ASTTrue ast) throws IVException;

    /**
     * Visit for ASTTypeDeclaration
     */
    public IVReturn visit(ASTTypeDeclaration ast) throws
        IVException;

    /**
     * Visit for ASTTypeId
     */
    public IVReturn visit(ASTTypeId ast) throws IVException;

    /**
     * Visit for ASTVarDec
     */
    public IVReturn visit(ASTVarDec ast) throws IVException;

    /**
     * Visit for ASTVars
     */
    public IVReturn visit(ASTVars ast) throws IVException;
}
```