# A TYPE SYSTEM FOR THE ERASMUS LANGUAGE

Nima Jafroodi

A thesis

in

The Department

of

Computer Science & Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

January 2008

*Your file Votre référence*
*ISBN: 978-0-494-40941-1*
*Our file Notre référence*
*ISBN: 978-0-494-40941-1*

# Abstract

A Type System for the Erasmus Language

Nima Jafroodi

Our objective is to construct a suitable *type system* for the Erasmus language by understanding the notion of *type* in programming languages, present a model of typed, polymorphic programming language that reflects recent research in type theory. This type system gives us a strong tool to explain the behavior of this language in a consistent framework by modeling features such as cells, processes, ports, protocols, messages, and message passing, which are the main heart of this programming language.

The Erasmus language belongs to process oriented programming languages which is being developed by Peter Grogono at Concordia University and Brian Shearing at The Software Factory in England. This language is mainly based on cells, processes and their interactions and like object oriented languages; it provides both a framework and a motivation for exploring the interaction among the concept of type, data abstraction, and polymorphism. We develop a $\lambda$-calculus-based model for this type system that allows us to explore these interactions in the simple setting.

The evolution of languages from untyped universes to monomorphic and polymorphic type systems is reviewed. Different type systems of different programming languages are discussed and the mechanism for polymorphism such as overloading, coercion, subtyping, protocols satisfaction, and parameterization are also described.

The typed $\lambda$-calculus is augmented to include binding of types by quantification as well as binding types by abstraction. This typed $\lambda$-calculus is augmented by *universal quantification* to model generic functions with type parameters, *bounded quantification* to provide explicit subtype parameters. In this way we obtain a simple and precise characterization of a powerful type system that includes abstract data types, parametric polymorphism, and subtyping relations in a single consistent framework.

This augmented typed $\lambda$-calculus is then used for the type system of the Erasmus language with which we are able to describe the features of this language, and a proof technique that will let us reason about the model.

# Acknowledgments

I would like to acknowledge Dr.Peter Grogono who directed me to wide range of resources. His advice and patience is much appreciated. I would also like to thank my parents and my sister who support me in every aspect of my life.

*Proofs of programs are too boring for the social process of mathematics to work. –Richard DeMillo, Richard Lipton, and Alan Perlis, 1979.*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Erasmus project is a process oriented programming language which is being developed by Peter Grogono at Concordia University and Brian Shearing at Software Factory in England. Like object oriented programming languages, Erasmus project provides both a framework and a motivation for exploring the interactions among the concept of type, data abstraction, and polymorphism. This process oriented programming language is mainly based on cells, processes and their interactions, and provides a full control over these interactions, which contrast with the object models in which an object doesn't provide a full control over the sequences in which method calls and events may hit an object.

The study of types and type systems has become one of the most important fields with major applications in the software engineering, language design, high performance compiler implementation and security. The formal type systems give us a mathematical model that can describe the features of programming languages; and a proof technique that will let us reason about the model.

## 1.1 Objectives and Goals

Our objective in this thesis is to first cover and describe the core topics and notations commonly used for describing type systems and then to construct and design a suitable type system for the Erasmus project that reflects recent researches in the type theory. This includes designing a formal type system, mathematical model, for checking type correctness and protocol satisfaction for the Erasmus project. Via this type system we will then be able to explain and describe all the behaviors of the Erasmus project in a consistent framework by modeling features such as cells, processes, ports, message passing and protocols which are the main heart of this process oriented language.

## 1.2 Organization

Chapter 2 will describe our problem and the motivation for the research presented in this thesis. We will then describe our proposal and its benefits.

In Chapter 3, we will explain some important and fundamental concepts commonly used for describing type systems, and we will seek to find the answers to the questions such as: What do we mean by the notion of type in programming languages? What is really a type system and what do we expect from it? What do we mean by the notions of component and their compatibility? What are typed and untyped languages, and what are their differences?

Furthermore, we will discuss the expected properties of a type system and how it can be formalized, as an example how a type system can be extended by the notion of subtyping and polymorphic approachs.

In Chapter 4, we will design and construct a suitable type system for the Erasmus language by reviewing a spectrum of simple types, pair and tuple types, array and map types, reference types, function types, and record types. We then formalize this informal type system by providing the appropriate judgments, which are formal assertions about the typing of programs, type rules, which are implications between judgments, and subtype relations for the types defined in our language. This type system will be then augmented by the appropriate type rules for statements such as conditional statements, arithmatic expressions and etc.

Chapter 5 introduces messages, protocols, ports, closures, and cells, which are the main heart of the Erasmus project, and we will then expand and formalize our previous type system with the appropriate type rules and subtype relations. Besides, we will provide some algorithms for checking protocol equality and protocol satisfaction, which enable us to describe the cells' interactions in a consistent framework.

Finally, Chapter 6 summarizes the achievements and the future works.

# Chapter 2

# Problem and Motivation and Proposal

Modern software engineering recognizes a broad range of *formal methods* for helping ensure that a system behaves correctly with respect to some specification, implicit or explicit, of its desired behavior. On one end of the spectrum are powerful frameworks such as Hoar logic, algebraic specification languages, model logics, and denotational semantics. These can be used to express very general correctness properties but are often cumbersome to use and demand a good deal of sophistication on the part of programmers. At the other end are techniques of much more modest power - modest enough that automatic checkers can be built into compilers, linkers, or program analyzers and thus be applied even by programmers unfamiliar with the underling theories. One well-known instance of this sort of *lightweight* formal methods is *model checkers*, tools that search for errors in finite-state systems such as chip design or communication protocols. Another is *run-time monitoring*, a collection of techniques that allow a system to detect, dynamically, when one of its components is

not behaving according to specification. But by far most popular and best established lightweight formal methods are *type systems* which is the central focus of this thesis.

Formal type systems are the mathematical model that can describe the features of programming languages by providing us with a technique for checking the type correctness of a program, and let us reason about the model. Therefore, this issue motivates us to understand the notions of types, and type systems.

We start from an informal typed language, and we design a formal type system for the Erasmus project to explain the behavior of this language in a consistent framework. Several type systems have been developed for proving that a system behaves correctly. We use typed-lambda calculus for the type system of the Erasmus project. This typed-lambda calculus includes the higher order functions which enable us to explain the behavior of this process oriented language in the simple settings. We then augment this typed-lambda calculus with the notion of *universal quantification* which enables us to model generic functions with the type parameters, we also augment this typed lambda calculus with the notion of *bounded quantification* to provide explicit subtype parameters to model ports, and closures in a consistent framework.

We also provide an algorithm for protocol satisfaction which includes reduction of infinite traces sets. Via this algorithm we are able then to explain the behaviors of the cells interactions.

# Chapter 3

# Theoretical Background

As the first step through our goals, we first look at some motivational issues, such as the need for plug-in compatible components and the different ways in which compatibility can be judge.

## 3.1 Components and Compatibility

The eventual economic of the object-oriented, process-oriented, and component-based software industry will depend on the ability to mix and match parts selected from different suppliers. Thus, the notion of component compatibility will play an important role in these languages. These components are categorized into two fundamental groups which are given below:

- The *client* (component user) who has to make certain assumptions about the way a component behaves in order to use it.
- The *server* (component provider) who has to build something in the way that can satisfy the expectations of its clients.

But how can one say that these two view points are compatible? The notion of type has been used to judge the compatibility in software. To better judge the compatibility between the components it is useful to categorize the compatibility into two fundamental ways:

- *Syntactic Compatibility*: the components provide all the expected operations.
- *Semantic Compatibility*: the components operations all behave in their expected ways.

As for the syntactic and semantic compatibility, a component must provide type names, function signatures, and interfaces, and it also must provide the logical axioms, state semantics, and the proofs to show that these operations behave in their expected ways.

These two view points of compatibility are so important that should be taken into the considerations while designing programming languages and their type systems. There are some spectacular examples failure due to the type-related software design faults, such as the Mars Climate Orbiter crash and the Ariane-5 launch disaster. These two examples clarify the differences between the two different kinds of incompatibility.

In the case of the Mars Climate Orbiter, the failure was due to the inadequate characterization of syntactic type, resulting in a confusion of metric and imperial units. Output from the spacecraft's guidance system was reinterpreted by the propulsion system in a different set of measurement units, resulting in an incorrect orbital insertion maneuver, leading to the crash. In the case of the Arian-5 disaster, the failure was due to the inadequate characterization of semantic type, in which the guidance system of the aircraft needlessly continued to perform its pre-launch self-calibration cycle in which the emission of the larger diagnostic code than expected caused an arithmetic overflow in the data conversion intended for propulsion system, which raised an exception terminating the guidance system.[16]

## 3.2 Types

To summarized so far, we've mentioned that the syntactic and semantic type compatibility of the components in the programming languages play an important role. For reasoning about the syntactic and semantic type compatibility of the components in a programming language we need an adequate definition of type.

What is a type?

Actually there is no unique definition of type. There are various definitions of type according to the perspective of different viewers.

- *Realist:* A type is a set of values.
- *Idealist:* A type is a conceptual entity whose values are accessible only through the interpretive filter of type.
- *Beginning Programmer:* Isn't a type a name for a set of values?
- *Intermediate Programmer:* A type is a set of values and operations.
- *Advanced Programmer:* A type is a way to classify values by their properties and behavior.
- *Algebraist:* A type is an algebra, a set of values and operations defined on values.
- *Type Checker:* Types are more practical than that, they are constraints on expressions to ensure compatibility between operators and their operand(s).

Our definition of types is also simple and close to the perspective of *advanced programmers*. To define the term type, it is useful to clarify the notion of *values*. A *value* is said to be anything that may be *evaluated, stored, incorporated in a data structure, passed as an argument* or *returned as a result.*

A program variable can accept a range of values during the execution of a program. This range of values is called *type* of that variable. As an example a variable $x$ of type *boolean* accepts only the values *true* or *false* during the running of program. Only if $x$ is variable of type *boolean* then the *boolean* expression $not(x)$ have a sensible meaning in every run of the program. Therefore, the notion of type is a way to classify values by their properties and behaviors.

## 3.3 Degrees Of Strictness And Sophistication

The notion of type has been used to judge the compatibility between components, but how strictly must a component matches into the interface in which it is plugged?

**Monomorphism** Conventional typed languages, such as Pascal which *is* a strongly typed language, are based on the idea that functions and procedures, and hence their operands, have a unique type that is, a variable can accept a value of the exactly the same type. This concept is known as *monomorphism* (literally, one form), and such languages are said to be *monomorphic* [6, 14, 16].

**Polymorphism** Monomorphic programming languages may be contrasted with *polymorphic* languages in which some values and variables may have more than one type. These polymorphic (literally, having many forms), allowing variables to receive values of more

than one type. Moreover, polymorphic functions are functions whose operands can have more than one type, and polymorphic types are types whose operations are applicable to values of more than one type [5, 6, 14, 16].

The polymorphic properties increase the generality of an interface which allows a wider choice of components to be substituted, which are said to satisfy the interface. A simple approach to interface satisfaction is *subtyping*. This is where an object of one type may safely be substituted where another type was expected. This involves no more than *coercing* the subtype object to a super type object and executing the super type's functions. Then the coerced object will behave exactly the same as the super type object. As an example; in C++ passing two *SmallInt* as an argument to the add function that accepts two Integers. The function is expecting two Integer values, but it could handle subtype of Integers and convert them. Also, a *simply-typed* first order calculus (with subtyping) is sufficient to explain this behavior.

It is useful in this place to distinguish between four kinds of polymorphism. These kinds of polymorphism are given in Figure 1 below.

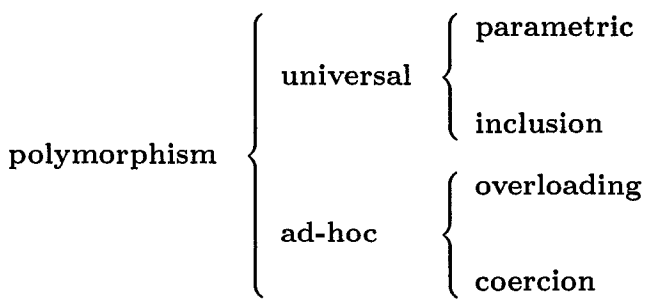


Figure 1: Varieties of polymorphism

*Parametric polymorphism* is obtained when a functions works uniformly on a range of types, and these types exhibit normally some common structure. *Inclusion polymorphism* models *subtypes* and *inheritance.* These two kinds of polymorphism are classified as the two major subcategories of *universal polymorphism* which will normally work on an infinite number of types (all types having a given common structure) [6, 17, 22].

We shall call the more complex, polymorphic approach *ad-hoc polymorphism* which is obtained when a function works, or appear to work, on several different types which may not exhibit a common structure, and may behave in unrelated ways for each type. This is where one type is replaced by another, which also systematically replaces the original functions with new ones appropriate to the new type. A *second order* calculus is sufficient to explain this behavior [5, 6, 16, 17].

In *overloading* the same variable name is used to denote different functions, and the context is used to decide which function is denoted by a particular instance of the name. A *coercion* is instead a semantic operation which is needed to convert an argument to the type expected by a function, in a situation which would otherwise result in a type error. Moreover, the functions that exhibit parametric polymorphism are called *generic functions* [6, 16, 17].

According to the materials mentioned above, there are three different degrees of sophistication when judging the compatibility of a component with respect to the expectations of an interface in programming languages:

- *Correspondence* the component is identical in type and its behavior exactly matches the expectations made of it when calls are made through the interface.

- *universal-polymorphism* the component is more specific type, but behaves exactly like

the more general expectations when calls are made through the interface.

- *ad-hoc-polymorphism* the component is a more specific type and behaves in ways that exceed the more general expectations when calls are made through the interface.

## 3.4 Type Systems

As with many terms shared by large communities, it is difficult to define "type systems" in a way that covers its informal usage by the programming language designers and implementers. A large percentage of errors in programs are due to the application of operations to the objects of incompatible types. Type systems have been developed to help the programmer to detect these errors. The fundamental purpose of a type system is to prevent occurrences of *execution errors* during the run time of a program. This informal definition motivates to study the type systems but needs more clarifications. When this property holds for all program fragments, we say that a language is *sound*, or more strictly, the type system of a language is *sound*. Moreover, when well developed, the type system provides conceptual tools with which to judge the adequacy of important aspect of language definitions such as components compatibility. There are lots of programming languages which are proved to be unsound such as $C$, $C++$. These programming languages allow a program to crash at the run time even though it is judged acceptable by the type-checker [6, 16].

As a formal definition of a type system we can say that: *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."* [14].

This definition has some crucial aspects. First, it identifies types as a tool for *reasoning* about the program. Second, this definition emphasizes on *classification* of terms according

to the properties of the values that will be computed during the run time.

## 3.5 Typed Languages and Untyped Languages

Languages that variables can be given types are called *typed languages*, and the languages that don't restrict types for their variables are called *untyped languages*. Untyped languages don't have types for their variables, or in another world they have a universal type that contains the whole range of values defined in the language [5]. Therefore, in an untyped language operations may be applied to inappropriate arguments, and the results may be: [5]

- A fixed arbitrary value
- A fault
- An exception
- Unspecified behavior

The untyped $\lambda$-calculus, pronounced "lambda calculus", is the extreme case of untyped languages that none of these fails never occur. In the untyped lambda calculus the operations are function applications and the values are functions, therefore the operations never fail [5, 6, 14].

The responsibility of the type system is to track the types of all variables and in general is to track all the expressions in the language. Thus, a type system accepts an expression if there is a type associated with it otherwise fails it.

To be more precise about the definition of a typed language and an untyped language, we can say that a language is a typed language by the virtue of the existence of a *type*

*system* for it, whether or not the types are in the syntax of the program. Otherwise is an untyped language. A typed language is said to be *explicitly typed* if the types are part of the syntax of the program, otherwise is said to be an *implicitly typed.* As an example languages such as ML, and Haskel are implicitly typed languages, and C++ and Java are said to be explicitly typed languages [5, 6].

## 3.6 Execution Errors

There are different kinds of execution errors according to their aspects. One of the most obvious aspects of execution errors is the occurrence of an unexpected behavior software fault, such as an illegal instruction fault or an illegal memory reference fault. However, there are some more execution errors that result in data corruption without any immediate symptoms. Also there are some software faults such as *division by zero* or *dereferencing nil* which is not allowed in most type systems.

Therefore to define our terminology about the errors, we categorize execution errors into two fundamental kinds:[5]

- *Trapped Errors:* the ones that cause the computation to stop immediately such as division by zero, or accessing an illegal address.

- *Untrapped Errors:* the ones that go unnoticed for a while and may cause unexpected behavior. Such as accessing a legal address *ex.* accessing data past the end of an array[1], or jumping to the wrong address *ex.* memory there may or may not represent an instruction stream.

[1] "Static elimination of array-bounds checking is a long standing goal for type system designers. In principle, the necessary mechanisms are well understood, but packaging them in a from that balances expressive power, predictability and tractability of type checking, and complexity of program annotations remains a significant challenge" [14].

## 3.7 Safe and Well Typed Programs

To summarize so far, we mentioned that it is helpful to distinguish between two kinds of execution errors which may occur during the running time of a program. The first ones were those that cause computations to stop immediately, trapped errors, and the other ones were those that go for a while and later may cause arbitrary behaviors, untrapped errors.

A program fragment is said to be ***safe*** if it doesn't let the untrapped errors to occur. Accordingly a language is said to be a *safe language* if all the program fragments were safe, otherwise, it is said to be an *unsafe language.* Therefore, safe languages rule out all the untrapped errors during the execution of the program, and it is one of the important facts of a language.[2]

Typed languages and untyped languages check safety in completely different ways. Typed languages perform safety at both before the execution of the program, at compile-time, *static checks*, and during the execution of the program, at the run-time enviroment, *dynamic checks.* Untyped languages perform safety at the run-time environment, *dynamic checks.*

Although, safety must be considered as one of the most important factors of a programming language, the type system of a language must also rule out some trapped errors as well as untrapped ones, such as division by zero or etc. Thus, for any given language, we designate a subset of all possible errors that may occur, and call it *forbidden errors.* The

[2] "Yet another point of view focuses on portability; it can be expressed by the slogan, "A safe language is completely defined by its programmer's manual." Let the definition of a language be the set of things the programmer needs to understand in order to predict the behavior of every program in the language. Then the manual for a language like *C* does not constitute a definition, since the behavior of some programs (e.g., ones involving unchecked array accesses or pointer arithmetic) cannot be predicted without knowing the details of how a particular *C* compiler lays out structures in memory, etc., and the same program may have quite different behaviors when executed by different compilers. By contrast the manual of Java, Scheme, and ML specify (with varying degrees of rigor) the exact behavior of all programs in the language. A well-typed program will yield the same results under any correct implementation of these languages" [14].

set of forbidden errors contains all the untrapped errors and a subset of trapped errors. A program is said to be *well-behaved* if it doesn't allow forbidden errors to occur, otherwise is said to be *not well-behaved* or equivalently *ill-behaved.* Moreover, a language is said to be *strongly checked* if all the programs have good behaviors, or it is said to be *not-checked* if not all the programs have good behavior.

Accrding to the materials mentioned above, strongly checked languages have the following properties:

- They must rule out all the untrapped errors.
- They also must rule out all the trapped errors defined in their set of forbidden errors.
- Other trapped errors may occur, and it is the responsibility of the programmer to take care of them.

The process of checking well behaved programs is called *type-checking*, and the algorithms that perform this checking is said to be a *type-checker.*

Typed languages ensure good behavior by performing static and dynamic checks to rule out all the untrapped and trapped errors defined in the forbidden set. These languages are said to be *statically checked.* Moreover, untyped languages enforce type checking at the run time environment, and they are said to be *dynamically checked* languages. A program that passes the type checker is said to be a *well-typed*; otherwise is said to be an *ill-typed.*

As it is obvious, a well-typed program is also safe, because it also rules out all the untrapped errors that may occur during the execution of the program. Even statically checked languages may perform some dynamically checks to obtain safety, for example array bounds must in general be tested dynamically, but these languages are still statically

checked, because the dynamic type tests are defined on the basis of the static type system. That is, the dynamic test for type equality are compatible with the algorithms that the type checker uses to determine type equality at compile time [5, 6, 14].

## 3.8 Characterization

So far, we've clarified the notions of safe and unsafe programming languages as well as typed and untyped languages. With these notions in mind, there are four different kinds of programming languages. As we can see in Figure 2, programming languages such as *ML* and *Java* are safe and typed languages, but *C* which is a typed languages is belonged to the unsafe languages. *LISP* and *Assembler* languages are the examples of the untyped languages with the difference that *LISP* is a safe language, but *Assemblers* are not.

| | Typed | Untyped |
|---|---|---|
| Safe | ML,Java | LISP |
| Unsafe | C | Assembler |

Figure 2: Safety and Typed

Studying these areas, arises these questions: Should programming languages be safe? and should languages be typed?

To answer these questions, we must take a look at the properties of being safe and being typed. Safety produces fail-stop behavior in case of execution errors, reducing debugging time. It also guarantees the integrity of run time structures, and therefore enables garbage collection [5]. Garbage collection [3] reduces the code size and the code development time.

[3]To be type-safe, a language must have garbage collection or otherwise restrict the allocation and de-allocation of memory. Specifically, it must not allow dangling pointers across structurally different types.

Finally, we can say that safety has emerged as a necessary foundation for system security, particularly for systems that load and run foreign code such as operating system kernels and web browsers.

Safety and safe languages have been studied for many years, but nowadays software engineers pay more attentions to the safety, because safety produces more security which is one of the main goals of the current languages.

Although, a lots of untyped programming languages are safe, but it is obvious that production codes in an untypes language can be maintained with great difficulty. The advantage of untyped languages is their *flexibility*. The programmer has complete control over how a data value is used but must assume full responsibility for detecting the application of operations to objects of incompatible type. Though, it has been proven that even an unsafe typed language is much better than safe but untyped languages [5, 6].

All we have discussed above let us to consider that a language should be both safe and typed. A language is a typed language by the virtue of the existence of the type system for it, thus a type system should be employed. Moreover, we mentioned that a language is well typed if it doesn't allow forbidden errors to occur, thus safety is an implied property of being well typed. But what should be the properties of a type system?

These are the basic properties of a type system:

- A type system should be *verifiable*; there should be an algorithm which ensures that a program is well-behaved. The type system must rule out all the forbidden errors

This is because if a typed language (like Pascal) required that allocated memory be explicitly released, and a dangling pointer still points to the old memory location, then a new data structure may be allocated in the same space with the slot the pointer refers to but point to a different type. For example, if the pointer initially points to a structure with an integer field, but in the new object a pointer field is allocated to the place of the integer, then the pointer field could be changed to anything by changing the value of the integer field (via dereferencing the dangling pointer. Because it is not specified what would happen when such a pointer is changed, the language is not type safe. Most type-safe languages satisfy these restrictions by using garbage collection to implement memory management.

before the execution of the program, and shouldn't allow some untrapped errors to occur accidentally.

- A type system should be *transparent*; a programmer should be able to predict if the program is well behaved, and if the program fails at the type check level then the failure reason must be evident.

- A type system should be *enforceable*; the type checking must be done statically as much as possible, otherwise the good behavior must be check dynamically.

Though, a type system which has these properties could be employed for programming languages. But still there is a problem according to the type system and that is, when a type system is defined how can we guarantee that well behaved program are really well behaved? Equivalently how can we be sure that the type rules of a program don't allow ill-behaviors to occur accidentally?

*Formal type systems* are the mathematical characterizations of the informal type system described in the programming language manuals. Formalizing a type system gives us tools to prove that a well typed program is really well behaved. If such a proof holds for all the well typed programs then the type system is said to be sound. Therefore, a formal type system is the key to prove the soundness of it.

## 3.9 Fomalizing Type Systems

There are some steps towards formalizing a type system which are given below:

First step of formalizing a type system is to describe its *syntax*, describing the syntax of a type system is to describe the *types* and *terms*. Types express the static knowledge about

the program, and terms (statements, expressions, and other program fragments) express the algorithmic behavior.

The second step towards formalizing a type system is to define the language *scoping rules*. Scoping rules of a language clarify the occurrences of the variables according to the areas in which they are defined. The scoping needed for a program is said to be static if the binding of identifiers are to be done at the compile time. The static binding of identifiers is said to be the *lexical scoping* otherwise is said to be *dynamically scoping.*

Scoping can be formally specified by defining a set of *free variables* of a program fragment which specifies how variables are bound in the declarations. When well developed, then the *substitution* of types and terms can be defined.

The third step in formalizing a type system is to define the *type rules* as a *has-type* relation, which is in the form of $M : A$, pronounced as term $M$ has type $A$. This could be regarded as set's membership relation $M \in A$. we can also assume that $M$ belongs to set $A$ if the term $M$ has type $A$. Moreover, some programming languages require the definition of *subtyping* relations in the form of $T <: T'$, pronounced type $T$ is subtype of type $T'$. It could be regarded as a *substitution* of different types but related. Also some programming languages need to define the *equal-type* relation in the form of $T = T'$, pronounced as type $T$ and $T'$ are equal.

Type rules cannot be formalized without defining a fundamental ingredient which is not defined in the syntax of the language. This fundamental element is *static type environment* which records the types of free variables during the processing of program fragments; they correspond closely to the symbol table of the compiler during the type checking phase. Type rules are always formulated with respect to the static type environment. For example in the

*has – type* relation $M : A$ is associated with the static type environment $\Gamma$ which contains the information about the free variables of $M$ and $A$. it is written in full as $\Gamma \vdash M : A$ meaning term $M$ has type $A$ in the environment $\Gamma$.

The final step in formalizing a type system is to define the semantics as a relation has-value between terms and collections of results, values.

## 3.10 The Language Of Formal Type Systems

To summarize so far, we mentioned that a language is said to be typed by the virtue of the existence of a type system for it, and we also mentioned that a formal type systems are the mathematical characterization of the informal type systems described in the language's manuals. Formal type systems are the keys to prove the soundness of a language.

### 3.10.1 Judgments

Type systems are described by a particular formalism. Actually the description of a type system starts with the collection of the formal context called *judgments*. A typical judgment is in the form of:

| *Judgment* | *Commentary* |
|---|---|
| $\Gamma \vdash \sigma$ | $\sigma$ is an assertion; the free variables of $\sigma$ are declared in $\Gamma$. |

In this context, $\Gamma$ is said to entails $\sigma$, where $\Gamma$ is the static type environment. The most important judgments to construct a type system are:

| | |
|---|---|
| $\Gamma \vdash \Diamond$ | $\Gamma$ is well-formed(i.e., it has been properly constructed) |
| $\Gamma \vdash M : A$ | M has type A in the type enviroment $\Gamma$ |

Note that any given judgment are either *valid* or *invalid.* As an example, the judgment "$\Gamma \vdash true : bool$" is a valid judgment and the judgment "$\Gamma \vdash true : Int$" is an invalid judgment. Validity of judgments formalizes the notion of well typed program.

### 3.10.2 Type Rules

Another step in formalizing a type system is to define the type rules of the language. Type rules assert the validity of a certain judgment on the basis of other judgments which are known to be valid. Typical type rules are in the form of:

$$\frac{\Gamma_1 \vdash \sigma_1 \ldots \Gamma_n \vdash \sigma_n}{\Gamma \vdash \sigma} \quad [\text{TYPE RULE NAME}]$$

Each type rule is written as a number of premises above the line and a conclusion below the line. The number of premises could be zero or more. A type rule is called an *axiom* if doesn't have any premises, and it is called a *theorem* otherwise. The most important axiom is:

$$\frac{}{\phi \vdash \Diamond} \quad [\text{ENV } \phi]$$

This axiom asserts that the empty enviroment $\phi$ is well-formed. If the promises hold valid then the conclusion is valid, and we say that from the premises we infer the conclusion. Thus any judgment can be proved to be valid or invalid with some type rules. As an example for theorems, Table 1 defines `Nat` by these type rules:

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash n : \mathtt{Nat}} \quad [\textsc{Val N} = 1,2,3,\ldots]$$

$$\frac{\Gamma \vdash M : \mathtt{Nat}, \Gamma \vdash N : \mathtt{Nat}}{\Gamma \vdash M + N : \mathtt{Nat}} \quad [\textsc{Val} +]$$

Table 1: Type rules for Nat

### 3.10.3 Type Derivations

A *derivation* in a given type system is a tree of judgments with the leaves at the top and the root in the bottom, where each judgment can be obtained immediately from the judgments above it by some rules of the type system. A judgment is *valid* if it can be derived from some valid jedgments by some given type rules. The problem of discovering the derivation for a term is called the *type inference problem.*

As an example of validity, it is easy to prove that the judgment "$\Gamma \vdash 1+2 : Nat$" is valid.

$$\frac{\dfrac{\Gamma \vdash \Diamond}{\Gamma \vdash 1 : \mathtt{Nat}} \ [\textsc{Val N}], \dfrac{\Gamma \vdash \Diamond}{\Gamma \vdash 2 : \mathtt{Nat}} \ [\textsc{Val N}]}{\Gamma \vdash 1 + 2 : \mathtt{Nat}} \ [\textsc{Val} +]$$

### 3.10.4 Subtyping

In the most component based languages, types are ordered by the *subtype* relation <:, and it is one of the approaches to interface satisfaction. Typically, an element of a type can be considered also as an element of any of its supertypes, thus allowing a value (object) to be used flexibly in many different typed contexts [17, 18].

Consider the type $A$ and $B$, we write $A <: B$ and say that $A$ is a *subtype* of $B$ and $B$ is said to be a *supertype* of $A$. The intuition is that, any element of type $A$ is also an element of type $B$ equivalently an element of type $A$ is acceptable in any context in which

an element of type $B$ is expected. This property of subtyping is called the *subsumption rule.*

The subtype relation is *partial order* meaning that it should be *reflexive*, *transitive*, and *antisymmetric.* These relations are described in the Table 2 below.

| | | |
|---|---|---|
| $\frac{}{T <: T}$ | [REFLEXIVE] | any type is a subtype of itself |
| $\frac{A <: B, B <: C}{A <: C}$ | [TRANSITIVE] | if $A <: B$ and $B <: C$ then A is also subtype of C |
| $\frac{A <: B, B <: A}{A = B}$ | [ANTISYMMETRIC] | if $A <: B$ and also $B <: A$ then they are equal |

Table 2: Subtype relation is partial order

The subtype relation defines relations over types of a programming language, and provides a wider choice of data to be substitute in a program.

### 3.10.5 Type Equality

As mentioned in the previous sections, programming languages need to define the relation *equal-type.*

Consider, the following two kinds of type:

$$type\ \mathrm{X} = int$$

$$type\ \mathrm{Y} = int$$

Some programming languages compare types with their given type names, in these languages type $X$, and $Y$ are not equal because of their distinct type names. In languages that types are compared by their names, the equal type relation is based on *name equivalences.*

On the other hand, some programming languages compare type by the means of comparing their structures. In these languages, type $X$ and $Y$ are said to be equal because of their equal structure. The equal type relation is called *structural equivalence* if the type system compares the types by their structural instead of their type names. Most programming languages use the benefits of both name equivalence and structure equivalences.

## 3.11 Different Kinds of Type Systems

In this section we will explain different methods for designing and constructing a type system. This includes explaining the properties of the type systems ranging from untyped-lambda calculus to the typed-lambda calculus such as the first order type systems, and the second order type systems.

### 3.11.1 Untyped Lambda Calculus

*Untyped lambda calculus* or equivalently *pure lambda calculus*, $\lambda$-calculus, is the method which is used in some functional programming languages such as *ML*, and *Haskell*. In the $\lambda$-calculus operations are functions and arguments are function applications, therefore operations never fail. "Untyped" actually means that there is only one type, and here the only type is the function type [5, 6, 14, 22].

Lambda calculus has been invented by *Alonzo Church* in 1920's, and is a formal system in which all computations are reduced to functions and function applications [7]. In mid 1960s, *Peter Landin* observed that all the complex programming languages can be understood by formulating it as a tiny *core calculus* capturing the language essential mechanism, together with a collection of convenient derived forms whose behavior is understood by translating

them into the core [11]. The core language that Landin used was lambda calculus.

The lambda calculus is one of the large numbers of the core languages that have been used for the same purposes. pi-calculus, $\pi$-calculus, is the other example of core languages which has been invented by *Milner, Parrow,* and *Walker* for defining the semantics of message-based concurrent languages [13]. The other example is *Abadi and Cardelli's object calculus* in 1996 for object oriented languages [1].

The reasons that we concentrate on introducing the lambda calculus is that, the lambda calculus can be enriched in variety of ways, and it gives us the basic ideas for introducing other type systems such as the *first order type system* and the *second order type system* which are used by the most programming languages.

*Procedural* or *functional* abstraction is an essential key for any programming languages. It is a wise idea to define a function, with or without parameters, to calculate something and call this function and instantiate its arguments as many times as we need in our program instead of writing the calculation each time. As an example, consider that we need to calculate this expression:

$$(7*6*5*4*3*2*1) + (6*5*4*3*2*1) + (5*4*3*2*1)$$

Therefore, instead of writing the repetitive expressions like above, we can define a function *Factorial* by this definition:

$$\text{Factorial(n)} = \text{if n} = 0 \text{ then 1 else n * Factorial(n-1)}$$

The intution is that for each n the function Factorial yields the factorial of n as a result. Thus if we write "$\lambda$n." as the shorthand for " the function that for each n, yields ...," then we redefine the definition of Factorial as:

Factorial = $\lambda$n. if if n = 0 then 1 else n * Factorial (n-1)

The $\lambda$-calculus embodies this kind of function definition and application in the purest possible form. In the $\lambda$-calculus everything is a function, arguments are functions and the results are also in the form of functions. In the $\lambda$-calculus there are only three kinds of terms. A variable $x$ by itself is a term, the abstraction of a variable $x$ in term $t$ is written $\lambda x.t$ is a term, and also the application of a term $M$ to term $N$, written $MN$, is a term.

In the $\lambda$-calculus an occurrence of the variable $x$ is said to be *bound* if it occurs in the body $t$ of the abstraction $\lambda x.t$. Moreover, $\lambda x$ is said to be a *binder*. An occurrence of $x$ is said to be *free* if it occurs in a position where it is not bound by an enclosing abstraction on $x$. Moreover, a term which doesn't have any free variables is said to be *closed*. A *closed* term is also called *combinator*.[14]

As an example, the occurrence of $x$ in $\lambda x.xy$ is bound and the occurrence of $x$ in $\lambda y.xy$ is free.

In the $\lambda$-calculus there are no built in constants or primitive operators. All the constant and operators are functions and the term computations are function applications. Each step in the computation is to rewrite an application whose left hand side is an abstraction and the right hand side is a term to be substitute for the bound variable defined in the abstraction. As an example $\lambda x.t\ (y)$ means substitute variable $x$ by $y$ in the body of $t$ which can be shown as:

$$\lambda x.t\ (y) \Rightarrow [\ x \mapsto y\ ]\ t$$

A term of the form $\lambda x.t$ (y) is said to be a *redex* "reducible expression" and the operation of rewriting a redex with the above method is called full-$\beta$-reduction.

### 3.11.2 First Order Type Systems

Most of the procedural based programming languages type systems, are called *first order type system.* The first order type systems lack the *type parameterizations* and *type abstractions* which are the features of the *second order type systems* [5, 6, 14]. This type system includes the higher order functions. *Pascal* and *Algol68* are the examples of the languages with the rich first order type systems while *FORTRAN* and *Algol60* have poor ones.

The first order type system also called system $F_1$, is similar to the untyped or pure lambda calculus. Functions in the first order type systems are in the form of $\lambda x : A.M$ in which $x$ is the bound variable of type $A$, and $M$ is the body of the function. The main difference between the first order type system and the pure lambda calculus is a type annotation for the bound variable. The step from $\lambda x.M$ to $\lambda x : A.M$ is typical of any progression from an untyped language to a typed language [6].

A type $A \rightarrow B$ is said to be a function type with the domain type $A$ to the range type $B$, or equivalently $A \rightarrow B$ is said to be a type of a function with the arguments of type $A$ and results of type $B$.

### 3.11.3 Second Order Type Systems

Many modern languages such as *Java* and *C#* uses the second order type system. These languages include constructions for *type parameterization*, and *type abstractions*, or both. Type parameters can be found in the module system of several languages, where a generic

module, class, or an interface is parameterized by a type to be supplied later. As an example $C\#$ uses type parameters at the class and interface level. $C++$ template are somehow similar to type parameters, but with the different methods. Polymorphic untyped languages such as $ML$, and $Haskell$ use type parameters pervasively just at the function level, as explained before. These advanced features can be so called *second order type systems* [5, 6, 14, 19].

Typically second order type systems extend first order type systems with the notion of type parameters. In the second order type systems, a new term $\lambda X.M$ is defined and asserts that the type identifier $X$ can be replaced by any arbitrary types, equivalently it is a function with the argument $X$ which stands for any arbitrary types with the program $M$ in which type $X$ may occur. As an example, consider the identity function *id* in the first order type systems, $\lambda x : A.x$, this function accept an identifier $x$ of type $A$ and returns it as a result. In the second order type system this identity function can be defined for all the types by this format, $\lambda X.\lambda x : X.x$. Therefore, one can instantiate the type variable $X$ with any arbitrary type. This method is one of the characterizations of polymorphic languages.

The second order type systems are also called system F2. Second order type systems extend first order type systems with the notions of *universally quantified types* [5] which enriches first order $\lambda$-calculus with parameterized types, [6], *Bounded Quantification* [6] which enriches first order $\lambda$-calculus by providing explicit subtype parameters, *Existential Quanification* [5, 6], and *F-Bounded Polymorphism* [3, 21] which enriches first order $\lambda$-calculus to provide a basis for typed polymorphic functions.

The type of a term $\lambda X.M$ is written $\forall X.A$ meaning that for all $X$, the body $M$ has type $A$. Let us consider the identity function provided before as an example, "$id = \lambda X.M$" where $M = \lambda x : X.x$, therefore the type of the function $id$ is $\forall X.X \rightarrow X$. In the F2 systems we drop the basic types, since we are now use type variables as the basic types. It turns out that we can construct any types with the system F2 by using the Existential types.

The scope rules of system F2 can be defined similar to the system F1. This means that in $\lambda x : A.M$, $\lambda x.$ binds $x$ inside $A$, and also in $\lambda X.M$ it binds type variable $X$ in the body $M$.

# Chapter 4

# Types In Erasmus

In this Chapter, we will construct the type system of the Erasmus project by describing the syntax of terms and types used in this language, and then we will formalize this type system by providing appropriate judgments, type rules, and subtyping rules according to the types we define.

Like any other programming languages, Erasmus language provides some basic types as well as pair and tuple types, function types, array and map types, record types, reference types, and facilitates for constructing an arbitrary number of record types.

As we mentioned in the previous chapter, one of the steps towards formalizing a type system is to describe the equal-type relation in the language, therefore the possible relationships between two types $T_1$ and $T_2$ in Erasmus language are:

$T_1 = T_2$ : $T_1$ and $T_2$ are *equal types*

$T_1 <: T_2$ : $T_1$ is a *subtype* of $T_2$

(no symbol) : $T_1$ and $T_2$ are unrelated

Erasmus language enforces type equality by using two methods. The first method is that all types except protocol types are compared by *name equivalence* that is; types are compared by their names not by their structures. One the other hand, Erasmus language enforces protocols equality by comparing their structures rather than their names which is briefly explained in the protocols section.

As mentioned before, one of steps towards formalizing a type system for a programming language is to describe the syntax of types and terms, and the other is to define some appropriate judgments with the appropriate type rules. The type system of the Erasmus project is based on the typed-lambda calculus. Therefore, the appropriate judgments needed for this type system are given in Table 3 below.

| | |
|---|---|
| $\Gamma \vdash \diamond$ | $\Gamma$ is well formed enviroment |
| $\Gamma \vdash A$ | A is well formed type in $\Gamma$ |
| $\Gamma \vdash M : A$ | M is well formed term of type A in $\Gamma$ |
| $\Gamma \vdash A <: B$ | A is a subtype of B in the enviroment $\Gamma$ |

Table 3: Judgments for Erasmus

## 4.1 Top and Bottom Types

It is useful in theory to have a bottom and top types in a programming language [14]. The name of the top type in Erasmus is *any*, with the following principal rule.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{any}} \quad \text{[T-ANY]}$$

$$\frac{\Gamma \vdash T}{\Gamma \vdash T <: \texttt{any}} \quad \text{[T-SUB-ANY]}$$

Table 4: Type Rules for Type Top

Moreover, the name of the bottom type in Erasmus language is *none*. The *none* type has no values. we can assume that `ext`[1] *none* $= \phi$.

## 4.2 Basic Types

Erasmus provides some basic types, these basic types are:

*BasicTypeName* = Void | Bool | Integer | Decimal | Float | Text.

These symbols are the keywords of the grammer, and following are their properties:

- The type `Void` has *void* as its value. It is useful to have such a type as a filler for uninteresting arguments and results.
- The type `Bool` has two values. These are *true* and *false* and are the keywords.
- The type `Integer` has integers in any range(not exceed the available memory) as its values.

[1] `ext T` is the set of all values having type `T`. As an example `ext Bool` $= \{true, false\}$

- The type `Decimal` is represented by values of the form $N \times 10^{-s}$ in which N is an Integer and s is a positive integer with limited range (e.g $0 \leq s \leq 255$).

- The type `Float` is represented by values with adequate range and precision for most purposes.

- The type `Text` has strings of ASCII characters as its values. The empty string is a value and there is no bound on the length of strings other than available memory.

Each basic types has a *default value*, and *null value*. The *null value* is used to indicate that no value is presented. The *default values* of basic types are shown below:

| Standard Type | Default Value |
|---|---|
| Bool | false |
| Integer | 0 |
| Decimal | 0.0 |
| Float | 0.0 |
| Text | "" |

The syntax of these basic types are given in Table 5, and Table 6 shows the appropriate type rules with subtyping relations for these basic types..

| T::= | types | | v::= | values |
|---|---|---|---|---|
| | Void | type Void | void | constant void |
| | Bool | Boolean type | true | constant true |
| | Integer | Integer type | false | constant false |
| | Decimal | Decimal type | nv | numeric values |
| | Float | Float type | tv | Text values |
| | Text | Text type | | |
| t::= | terms | | | |
| | void | constant void | true | constant true |
| | false | constant false | | |
| | x | identifier | | |

Table 5: Syntax of Basic Types In Erasmus

| | |
|---|---|
| $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Void}}$ [T-VOID] | $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{void} : \texttt{Void}}$ [T-VAL VOID] |
| $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Bool}}$ [T-BOOL] | $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{true} : \texttt{Bool}}$ [T-VAL TRUE] |
| $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Integer}}$ [T-INTEGER] | $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{false} : \texttt{Bool}}$ [T-VAL FALSE] |
| $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Decimal}}$ [T-DEC] | $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Integer} <: \texttt{Decimal}}$ [T-SUB INT-DEC] |
| $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Float}}$ [T-FLOAT] | $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Integer} <: \texttt{Float}}$ [T-SUB INT-FLOAT] |
| $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Text}}$ [T-TEXT] | $\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Decimal} <: \texttt{Float}}$ [T-SUB FLOAT-DEC] |

Table 6: Type Rules for Basic types in Erasmus

## 4.3 Function Types

Since the type system of the Erasmus language is mainly based on the typed-lambda calculus, the most interesting types are functions. The syntax $\lambda x : T.e$ is equal to a function with the body $e$ and with the function parameter $x$ of a specific type $T$. The lambda abstraction automatically bounds the variable $x$ in the body $e$.

Moreover, Erasmus langauge provides some built-in functions which are the keywords of the grammar. These built-in functions are:

$$\textit{FunctionName} = \texttt{int} \mid \texttt{float} \mid \texttt{text}$$

These functions accept an argument and if applicable convert them into the type of their function name. For example int(e) converts $e$ into integer, and if successful, returns the result. Table 7 shows the syntax of function types and the appropriate type rules are given in Table 8. (These built-in functions can also be described by using the *Bounded Universal Quantification* which will be introduced later.)

| | | | |
|---|---|---|---|
| T::= | | types | |
| | $T_1 \rightarrow T_2$ | | function type |
| t::= | | terms | |
| | $\lambda x : T.t$ | | function abstraction |
| | t t | | function application |
| | x | | identifier |
| | int(t) | | built-in function |
| | float(t) | | built-in function |
| | text(t) | | built-in function |

Table 7: Syntax Of Function Types In Erasmus

Note that type rule (T-Env) is used to extend an environment $\Gamma$ to a larger environment $\Gamma, x : A$ , provided that $A$ is a valid type in $\Gamma$ that is, we are careful to keep variables and types distinct in our enviroments. The type rule $\Gamma, x : A \vdash M : B$ means that the term $M$ has type $B$ in our static enviroment $\Gamma$ with the assumption that the variable $x$ has type $A$.

The type rule (T-Sub-Fun) indicates the subtyping relation between fucntion types and needs more explanations. For a substitute function $F_1 : D_1 \rightarrow R_1$ to behave exactly like the original function $F_2 : D_2 \rightarrow R_2$ the following conditions must hold true:

- $F_1$ must be able to handle at least as many argument values as $F_2$ could accept; we express this as a constraint on the domains (argument types): $D_2 <: D_1$ and
- $F_1$ must deliver a result that contains no more values than the result of $F_1$ expected; we express this as a constraint on the codomains (result types): $R_1 <: R_2$.

Thus, the codomain is also a subtype, but the domain is a supertype. For this reason, we sometimes say that the domains are *contravariant* (they are ordered in the opposite direction) and the codomains are *covariant* (they are ordered in the same direction) with respect to the subtyping relationship between the functions [17].

One of the most important responsibilities of a type system is to ensure that no forbidden errors will ever occur. Therefore, it is useful in this place to introduce a new term called *wrong* and augment the operations semantics with the rules that explicitly generate wrong in all the situations where the present semantics get stuck [14]. One can assume wrong as the set of all run-time errors. These type rules are shown by the letter E within our type system. As an example the rule "E-intWrong" in the Table 8 indicates that the built-in function "int" doesn't accept a boolean value or a Text value which is not an integer as

$$\frac{}{\Gamma \vdash \diamond} \quad \text{[T-ENV]}$$

$$\frac{\Gamma \vdash \mathtt{A}, x \notin dom(\Gamma)}{\Gamma, x : \mathtt{A} \vdash \diamond} \quad \text{[T-ENV X]}$$

$$\frac{\Gamma, x : \mathtt{A} \vdash \diamond}{\Gamma, x : \mathtt{A} \vdash x : \mathtt{A}} \quad \text{[T-VAL X]}$$

$$\frac{\Gamma \vdash \mathtt{A}, \Gamma \vdash \mathtt{B}}{\Gamma \vdash \mathtt{A} \rightarrow \mathtt{B}} \quad \text{[T-FUNCTION]}$$

$$\frac{\Gamma, x : \mathtt{A} \vdash \mathtt{M} \rightarrow \mathtt{A}}{\Gamma \vdash \lambda x : \mathtt{A}.\mathtt{M} : \mathtt{A} \rightarrow \mathtt{B}} \quad \text{[T-VAL FUNCTION]}$$

$$\frac{\Gamma \vdash \mathtt{M} \ : \ \mathtt{A} \rightarrow \mathtt{B}, \Gamma \vdash \mathtt{N} \ : \ \mathtt{A}}{\Gamma \vdash \mathtt{M\ N} \ : \ \mathtt{B}} \quad \text{[T-FUN APPL]}$$

$$\frac{\Gamma \vdash \mathtt{D}_2 <: \mathtt{D}_1, \Gamma \vdash \mathtt{R}_1 <: \mathtt{R}_2}{\Gamma \vdash \mathtt{D}_1 \rightarrow \mathtt{R}_1 <: \mathtt{D}_2 \rightarrow \mathtt{R}_2} \quad \text{[T-SUB-FUN]}$$

$$\frac{\Gamma \vdash \mathtt{E} \ : \ \mathtt{A}, \mathtt{A} \in \{\mathtt{Integer}, \ \mathtt{Decimal}, \ \mathtt{Float}, \ \mathtt{Text}\}}{\Gamma \vdash \mathtt{int(E)} \ : \ \mathtt{A} \rightarrow \mathtt{Integer}} \quad \text{[T-FUNC INT]}$$

$$\frac{\Gamma \vdash \mathtt{E} \ : \ \mathtt{A}, \ \mathtt{A} \in \{\mathtt{Integer}, \ \mathtt{Decimal}, \ \mathtt{Float}, \ \mathtt{Text}\}}{\Gamma \vdash \mathtt{float(E)} \ : \ \mathtt{A} \rightarrow \mathtt{Float}} \quad \text{[T-FUNC FLOAT]}$$

$$\frac{\Gamma \vdash \mathtt{E} \ : \ \mathtt{A}, \ \mathtt{A} \in \{\mathtt{Decimal}, \ \mathtt{Float}, \ \mathtt{Integer}, \ \mathtt{Bool}, \ \mathtt{Text}\}}{\Gamma \vdash \mathtt{text(E)} \ : \ \mathtt{A} \rightarrow \mathtt{Text}} \quad \text{[T-FUNC TEXT]}$$

$$\frac{\Gamma \vdash \mathtt{M} \ : \ \mathtt{Bool}, \ \mathtt{Bad(Text)}}{\mathtt{int(M)} \rightarrow \mathtt{wrong}} \quad \text{[E-INTWRONG]}$$

$$\frac{\Gamma \vdash \mathtt{M} \ : \ \mathtt{Bool}, \mathtt{Bad(Text)}}{\mathtt{float(M)} \rightarrow \mathtt{wrong}} \quad \text{[E-FLOATWRONG]}$$

Table 8: Type Rules For Function Types In Erasmus

its argument. Therefore passing any bad arguments to this function will be evaluated to wrong.

From now on in this article, we omit the wrong evaluation rules from our type system and we will assume that all the situations which are not defined in our type system will be evaluated to wrong.

## 4.4 Reference Type

*Name binding* in programming languages is the association of values with identifiers, but in most programming languages the mechanisms for name binding and those for assignment are kept separate. As an example we can have variable $x$ whose value is the number 5, or a variable $y$ whose value is a *reference*, or a *pointer* to a mutable cell whose current content is 5. What is the difference? The difference is that we can add $x$ to another number, but not assign to it. We can use $y$ to assign a new value to the cell that it points to by writing $y := 7$, but we can not use it directly as an argument to a function. Instead we can explicitly dereference it to obtain its value [14].

In Erasmus as well as most of the languages every variable name refers to a mutable cell, and the operation of dereferencing a variable to obtain its current contents is implicit. Strictly speaking, most variables of type $T$ should actually be thought of as pointers to cells holding values of type `ref` $T$, reflecting the fact that the contents of a variable can be either a proper value or the special vale `Null`.

Therefore, an element of $ref(A)$ is a mutable cell containing an element of type $A$. A new cell can be allocated by (Val Ref), and explicitly dereferenced by (Val Deref). Reference

types are useful in assignment and passing arguments by reference. As an example consider the assignment of two variables in the form of $x := y$, in this case $y$ is implicitly dereferenced to obtain its value, then this value will be copied to the address associated with $x$ which is not dereferenced.

Thus, we are justified to define a new type called *reference type* and augment it into our existing type system. Table 9 shows the appropriate type rules for the reference types.

$$\frac{\Gamma \vdash \texttt{A}}{\Gamma \vdash \texttt{ref A}} \quad \text{[T-REFERENCE]}$$

$$\frac{\Gamma \vdash \texttt{M : A}}{\Gamma \vdash \texttt{ref M : ref A}} \quad \text{[T-VAL REF]}$$

$$\frac{\Gamma \vdash \texttt{M : ref A}}{\Gamma \vdash \texttt{deref M : A}} \quad \text{[T-VAL DEREF]}$$

Table 9: Type Rules for Erasmus

## 4.5 Pair and Tuple Types

An immediately useful construction which we do not yet have is the notion of a pair of values, $< a, b >$, possibly taken from different types $A$ and $B$. The type of a pair is also known as a *product type*, or *Cartesian product*, since there are $A \times B$ possible paring of element $a \in A$ and $b \in B$. Pair types are useful in describing the functions with multi arguments.

Tuple types are the extended case of pair types, or equivalently pair types are the special case of tuples. Instead of having two elements in the pair types, we can have as many elements as needed in tuple types.

Adding pairs and tuples to the simply typed lambda-calculus involves adding two new forms of pairing, witten $t =< t_1, t_2, \ldots, t_n >$, and projection, written $\pi_1(t)$ for the first projection from $t$ and $\pi_n(t)$ for the n'th projection plus one new type constructor, $T_1 \times \ldots \times T_n$. The syntax of pairs and tuple types are given in Table 10 below.

| T::= | types | | v::= | values | |
|---|---|---|---|---|---|
| $T_1 \times T_2$ | | Pair type | $< v, v >$ | | pair value |
| $T_1 \times T_2 \times \ldots \times T_n$ | | Tuple type | $< \underbrace{v, \ldots, v}_{n \geq 1} >$ | | tuple value |
| t::= | terms | | | | |
| $< t, t >$ | | pair | | | |
| $< \underbrace{t, \ldots, t}_{n \geq 1} >$ | | tuple | | | |
| $\pi_i(t)$ | | projection | | | |

Table 10: Syntax of Pairs and Tuples In Erasmus

And the appropriate type rules with the subtyping relations for these types are given in Table 11 below.

$$\frac{\Gamma \vdash \mathtt{m} : \mathtt{M}, \Gamma \vdash \mathtt{n} : \mathtt{N}}{\Gamma \vdash \mathtt{<n,\ m>} : \mathtt{N} \times \mathtt{M}} \quad \text{[T-Pair]}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{N} \times \mathtt{M}}{\Gamma \vdash \pi_1(\mathtt{e}) : \mathtt{N}, \Gamma \vdash \pi_2(\mathtt{e}) : \mathtt{M}} \quad \text{[T-Pair Projections]}$$

$$\frac{\Gamma \vdash x_i : T_i}{\Gamma \vdash < x_1, \ldots, x_n > : T_1 \times \ldots \times T_n} \quad \text{[T-Tuple]}$$

$$\frac{\Gamma \vdash \mathtt{e} : T_1 \times \ldots \times T_n}{\Gamma \vdash \pi_1(e) : T_1, \ldots, \pi_n(e) : T_n} \quad \text{[T-Val Tuple]}$$

$$\frac{\Gamma \vdash \mathtt{N} <: \mathtt{N'}, \Gamma \vdash \mathtt{M} <: \mathtt{M'}}{\Gamma \vdash \mathtt{(e : N \times M)} <: \mathtt{(e' : N' \times M')}} \quad \text{[T-Sub-Pair]}$$

$$\frac{\Gamma \vdash T_i <: T_i'}{\Gamma \vdash \mathtt{(e :}\ T_1 \times \ldots \times T_n) <: \mathtt{(e' :}\ T_1' \times \ldots \times T_n')} \quad \text{[T-Sub Tuple]}$$

Table 11: Type Rules and Subtyping Rules For Pair and Tuple Types

## 4.6 Map And Array Type

The types described in this section are defined by the programmers from a small set of construction rules. *maps* are used to construct tables which can be accessed by indexing. As an example, *arrays* are the map type in which index is an integer. A map type is defined in terms of two other types: a *domain type* or *index type*, and a *range type*. In Erasmus maps are shown within this syntax "*Domain Type* `indexes` *Range Type*".

In order to explain the type rules for maps, we need a second order type system, system $F_2$. Second order type system extend first-order type systems with the notion of *type parameters*. A new kind of term written $\lambda X.M$, indicates a programme $M$ that is parameterized with respect to a type variable $X$ that stands for an arbitrary type.

Corresponding to the new term $\lambda X.M$ we need to define a new type which is known as *universally quantified types* [6, 20]. The type of a term $\lambda X.M$ is written $\forall X.A$, meaning that for all types $X$, the body $M$ has the type $A$(here $M$ and $A$ may contain occurrences of $X$).

Free variables for $F_2$ types and terms can be defined in the usual fashion, that is, $\forall X.A$ binds $X$ in $A$ and $\lambda X.M$ binds $X$ in $M$.

Following table augments the appropriate syntax of the system $F_2$ into our type system.

| | | | |
|---|---|---|---|
| T::= | | types | |
| | X | | type variable |
| | ∀X.A | | universally quantified type |
| t::= | | terms | |
| | x | | variable |
| | λX.M | | polymorphic abstraction |
| | M A | | type instantiation |

Table 12: Syntax of $F_2$

The appropriate type rules of $F_2$ are given in Table 13 below.

$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \forall X.A} \quad \text{[T-FORALL]}$$

$$\frac{\Gamma, X \vdash M : A}{\Gamma \vdash \lambda X.M : \forall X.A} \quad \text{[T-VAL FUNCTION } F_2\text{]}$$

$$\frac{\Gamma, X \vdash M : \forall X.A, \ \Gamma \vdash B}{\Gamma \vdash MB : [B/X]A} \quad \text{[T-VAL APPL } F_2\text{]}$$

Table 13: Type Rules of $F_2$

Note that the type rule (T-Val Appl $F_2$) instantiates a polymorphic abstraction to a

given type, where $[B/X]A$ is the substitution of $B$ for all the free occurrences of $X$ in $A$. As an example, consider an id function which has the type $\forall X.X \rightarrow X$, then by (T-Val Appl $F_2$) we have that id A has type $[A/X](X \rightarrow X) \equiv A \rightarrow A$.

Now let's get back to the map and array types. As mentioned above the syntax of maps in Erasmus is "*Domain Type* **indexes** *Range Type.* This type could be explained by the universally quantified types, that is:

$$T_1 \texttt{ indexes } T_2 \;=\; \lambda D.(\lambda R.D \rightarrow R) : \forall D.(\forall R.D \rightarrow R) \;\equiv\; D \mapsto R$$

As an example Array(T) could be explained by:

$$\text{Array(T)} \;=\; \lambda x : T.(\texttt{Integer} \rightarrow T) : \texttt{Integer} \mapsto T.$$

Note that maps are the first class values, they are legal values which can be passed and returned from functions and stored in data structures. Therefore, the indexing operator returns a reference type meaning that it is an l-value and could be used in assignments. Therefore, we use the symbol "$\mapsto$" to make a distinction between map tyes and function types [14]. Table 14 below contains the formal definitions of map types.

| | | | |
|---|---|---|---|
| T::= | | types | |
| | T = D $\mapsto$ R | | Map type |
| t::= | | terms | |
| | T indexes T | | map abstraction |
| | Integer indexes T | | array abstraction |
| | t[t] | | indexing operator |

Table 14: Syntax Of Map And Array In Erasmus

And the appropriate type rules for map and array types are given in Table 15 below.

$$\frac{\Gamma \vdash \texttt{D,R}}{\Gamma \vdash \texttt{D} \mapsto \texttt{R}} \quad \text{[T-MAP]}$$

$$\frac{\Gamma \vdash \texttt{Integer}, \Gamma \vdash \texttt{R}}{\Gamma \vdash \texttt{Integer} \mapsto \texttt{R}} \quad \text{[T-ARRAY]}$$

$$\frac{\Gamma \vdash \texttt{M} : \texttt{D} \mapsto \texttt{R}}{\Gamma, \texttt{x} : \texttt{D} \vdash \texttt{M[x]} : \texttt{ref R}} \quad \text{[T-VAL MAP]}$$

$$\frac{\Gamma \vdash \texttt{D}_2 <: \texttt{D}_1, \Gamma \vdash \texttt{R}_1 <: \texttt{R}_2}{\Gamma \vdash \texttt{D}_1 \mapsto \texttt{R}_1 <: \texttt{D}_2 \mapsto \texttt{R}_2} \quad \text{[T-SUB MAP]}$$

Table 15: Type Rules For Map And Array Types In Erasmus

## 4.7 Record Type

A record is a set of finite pairs; consider pairs as mappings from labels to values, for example $\{< name, "John" >, < age, 25 >, < StudentId, "5746" >, \ldots\}$ then we can say that a record is a set of finite mappings from labels to values. Since a record has this clear, we are justified to introduce a new syntax.

*ParentType* = '[' {*FieldDeclaration*}, ']'.

*ChildType* = TypeIdentifier '[' {*FieldDeclaration*}, ']'.

*FieldDeclaration* = [ transient ] FieldIdentifier ':' *TypeExpression.*

Table 16 shows the syntax and Table 17 shows the type rules with subtyping rules for the record types.

| | | | |
|---|---|---|---|
| T::= | | types | |
| | $\{\alpha_1 : T_1, \ldots, \alpha_i : T_i\}$ | | record type |
| t::= | | terms | |
| | $\{\alpha_1 = t_1, \ldots, \alpha_i = t_i\}$ | | record |
| | $t.\alpha_i$ | | projection |
| v::= | | value | |
| | $\{\alpha_1 = v_1, \ldots, \alpha_i = v_i\}$ | | record value |

Table 16: Syntax for Records

$$\frac{\Gamma \vdash \alpha_i : A, \Gamma \vdash \beta_i : T_i}{\Gamma \vdash \{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\} : \{\alpha_1 : T_1, \ldots, \alpha_n : T_n\}} \quad [\textsc{Type Record}]$$

$$\frac{\Gamma \vdash R : \{\alpha_1 : T_1, \ldots, \alpha_n : T_n\}}{\Gamma \vdash R.\alpha_i : T_i} \quad [\textsc{Val Record}]$$

$$\frac{\Gamma \vdash P : \{\alpha_1 : T_1, \ldots, \alpha_n : T_n\}, C : P\,\{\alpha_{n+1} : T_{n+1}, \ldots, \alpha_{n+m} : T_{n+m}\}}{\Gamma \vdash C : \{\alpha_1 : T_1, \ldots, \alpha_{n+m} : T_{n+m}\}} \quad [\textsc{Child-Rec}]$$

$$\frac{\Gamma \vdash P : \{\alpha_1 : T_1, \ldots, \alpha_n : T_n\}, \Gamma \vdash C : P\,\{\alpha_{n+1} : T_{n+1}, \ldots, \alpha_{n+m} : T_{n+m}\}}{\Gamma \vdash C <: P} \quad [\textsc{Sub Child}]$$

Table 17: Type Rules for Erasmus

The type rule "Type Record" indicates that if there is a set containing some mappings from labels to values then this set has the type record. Moreover, the type rule "Val Record" introduce the operator "." on the records which gives us a tool for describing the type of the record eliminations. As it is obvious in the syntax of records, a record type may be a parent type or a child type. One can assume records as equal to sets, thus every operation which is acceptable on the parents are also acceptable on their children. The type rule "Sub Child" indicates a subtyping relation between record types that is, the type of a child record is always a subtype of its parent.

## 4.8 Statements

Statements include declarations and actions. A sequence is a series of statements that are executed consecutively. One of the steps in formalizing a type system is to define its *scoping rules* of the language.

A sequence defines a scope. The variable introduced by a declaration is visible from the point of the declaration to the end of the sequence containing the declaration. Following the usual conventions for nested scopes, outer scopes are accessible from inner scopes.

$$
\begin{aligned}
\textit{Sequence} &= \textit{Statement};\\
\textit{Statement} &= \text{skip}\\
&\mid \text{exit}\\
&\mid \text{until } \textit{Expression}\\
&\mid \text{while } \textit{Expression}\\
&\mid \textit{Declaration}\\
&\mid \textit{Instantiation}\\
&\mid \textit{Expressions with Binary Operators}\\
&\mid \textit{Conditional}\\
&\mid \textit{Loop}\\
&\mid \textit{Select}
\end{aligned}
$$

Following is the appropriate type rules for introducing sequences.

$$\frac{\Gamma \vdash S_1, S_2 : \texttt{Void}}{\Gamma \vdash (S_1; S_2) : \texttt{Void}} \quad \text{[T-SEQUENCES]}$$

Table 18: Type Rule For Sequences

## 4.9 Variable Declaration

The syntax for the variables declarations in Erasmus project are given in Table 19 below. If the expression contains a colon and a type, it is a *declaration*, but if it contains an assignment operator and an expression and a colon and a type then it is an *initialized declaration*. The type rules for a declaration and an initialized declaration are given in

Table 20.

| | | |
|---|---|---|
| *VarDeclaration* | = | VariableName [ ":" [*Mode*] *Type* ][":=" Rvalue]. |
| *Mode* | = | copy \| share \| alias |

| | | |
|---|---|---|
| t::= | | terms |
| | x | identifier |
| | t : T | declaration |
| | t : T := v | initialized declaration |
| | t : M T | declaration with mode |
| M::= | | mode |
| | copy | keyword copy |
| | share | keyword share |
| | alias | keyword alias |

Table 19: Syntax For Variable Declaration

$$\frac{\Gamma \vdash A,\ x \notin dom(\Gamma)}{\Gamma, x{:}A \vdash \diamond}\ [\text{T-ENV x}] \qquad \frac{\Gamma, x{:}A \vdash \diamond}{\Gamma, x{:}A \vdash x{:}A}\ [\text{T-DECL}]$$

$$\frac{\Gamma, x{:}A \vdash \diamond}{\Gamma, x{:}A \vdash (x:\ \ copy\ A)\ :\ Void}\ [\text{T-DECL 1}]$$

$$\frac{\Gamma, x{:}A \vdash \diamond}{\Gamma, x{:}A \vdash (x:\ \ share\ A):\ Void}\ [\text{T-DECL 2}]$$

$$\frac{\Gamma, x{:}A \vdash \diamond}{\Gamma, x{:}A \vdash (x:\ \ alias\ A)\ :\ Void}\ [\text{T-DECL 3}]$$

$$\frac{\Gamma, x{:}A \vdash M\ :\ B, \Gamma \vdash B <: A}{\Gamma, x{:}A \vdash (x\ :\ \ A\ :=\ M)\ :\ Void}\ [\text{INIT DECL}]$$

Table 20: Type Rules for Declarations in Erasmus

The type rule "Env x" expands our type system and the type rule "Decl" indicates that variables declarations are the well formed terms in our static environment. Same as before, we are careful to keep variables and types distinct in our enviroments.

## 4.10 Expressions with Binary Operators

Binary operators act on two operands in an expression. The binary operators in the Erasmus project are:

- `Multiplicative Operators`
  - Multiplication (`*`)
  - Division (`/`, `div`)
  - Modulus (`mod`).
- `Additive Operators`
  - Addition (`+`)
  - Subtraction (`-`)
- `Relational and Equality Operators`
  - Less than (`<`)
  - Greater than (`>`)
  - Less than or equal to (`<=`)
  - Greater than or equal to (`>=`)
  - Equal to (`=`)
  - Not equal to (`!=`)
- `Logical Operators`
  - Logical And (`and`)

- Logical Or (`or`)
- Logical Not (`not`)

- `Assignment operators`
  - Assignment (`=`)
  - Addition assignment (`+=`)
  - Subtraction assignment (`-=`)
  - Multiplication assignment (`*=`)
  - Division assignment (`/=`)
  - Modulus assignment (`%=`)

Each binary operators acts on two operands, that is:

``` ``Left Operand'' ``Operator'' ``Right Operand''. ```

### 4.10.1 Multiplicative and Additive Operators

It is useful in this place to indicate that the infix notation $x + y$ is an abbreviation for the functional notation $+\ (x)\ (y)$. The symbol $+$ should be viewed as an abbreviation for a pure lambda calculus expression for numbers. This is same for other binary operators as well.

$$+ \ : \ \lambda(x : X).\lambda(y : Y).(x + y)$$

One should pay attention that these operators accept operands from different types. As an example the operator $+$ accepts arguments of type `Float, Decimal, Integer, Text`, and the operator $*$ accepts arguments of type `Float, Decimal, Integer`.
```

Now consider the following example for the operator +.

$$3 \quad + \quad 4$$

$$3.0 \quad + \quad 4$$

$$3 \quad + \quad 4.0$$

$$3.0 \quad + \quad 4.0$$

Here the ad-hoc polymorphism of + can be explained in one of the following ways:

- The operator + has four overloaded meanings, one for each of the four combination of argument types.
- The operator + has two overloaded meanings, corresponding to `Integer` and `Float` addition. When one of the argument is of type `Integer` and the other is of type `Float` then the `Integer` argument is coerced to the type `Float`.
- The operator + is only defined for `Float` addition, and other types are coerced into the type `Float`.

In the Erasmus type system we use *bounded universal quantification*, *coercion*, and *overloading* to explain the behavior of these generic functions.

*Bounded universally quantification* is some how same as universally quantified types with subtyping. A new kind of term written $\lambda[X <: Y].M$, indicates a programme $M$ that is parameterized with respect to a type variable $X$ which is subtype of type variable $Y$. Note that $X$ and $Y$ stand for an arbitrary types.

Therefore, corresponding to our new term, the type of a term $\lambda[X <: Y]$.M is written $\forall X <: Y.A$ meaning that for all types $X$ which are all subtypes of type $Y$, the body $M$ has

the type $A$. Here $X$ ranges over all subtypes of $Y$ in the scope $M$.

Table 21 given below augments the appropriate syntax of *bounded universally quantification* into our type system.

| | | | |
|---|---|---|---|
| T::= | | types | |
| | X | | type variable |
| | ∀X<:Y.A | | bounded universally quantified type |
| t::= | | terms | |
| | x | | variable |
| | λ[X <: Y].M | | polymorphic abstraction |
| | M A | | type instantiation |

Table 21: Syntax of $F_2$ with Bounded Universally Quantified Types

The appropriate type rules for bounded universal quantifiers are given in Table 22 below.

$$\frac{\Gamma, X <: A \vdash B}{\Gamma \vdash \forall X <: A.B} \quad \text{[T-FORALL]}$$

$$\frac{\Gamma, X <: A \vdash M : B}{\Gamma \vdash \lambda X <: A.M : \forall X <: A.B} \quad \text{[T- VAL BQT1]}$$

$$\frac{\Gamma \vdash M : \forall X <: A.B, \Gamma \vdash A' <: A}{\Gamma \vdash MA' : [A'/X]B} \quad \text{[T- VAL BQT2]}$$

$$\frac{\Gamma \vdash A' <: A, \Gamma, X <: A' \vdash B <: B'}{\Gamma \vdash (\forall X <: A.B) <: (\forall X <: A'.B'))} \quad \text{[T-SUB FORALL]}$$

Table 22: Syntax of $F_2$ with Bounded Universally Quantified Types

According to the bounded universally quantification the type rules for multiplicative and additive operators are given in Table 23 below.

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}, \Gamma \vdash X <: Y}{\Gamma, x : X, y : Y \vdash (x + y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow Y} \quad \text{[T-OPRT-NUMERIC +]}$$

$$\frac{\Gamma, x : \texttt{Text} \vdash x : \texttt{Text} \;\; \Gamma, y : \texttt{Text} \vdash y : \texttt{Text}}{\Gamma \vdash (x + y) : \texttt{Text}} \quad \text{[T-OPRT-TEXT +]}$$

$$\frac{\Gamma \vdash X <: \texttt{Float} \;, \Gamma \vdash Y <: \texttt{Float}, \Gamma \vdash X <: Y}{\Gamma, x : X, y : Y \vdash (x - y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Y}} \quad \text{[T-OPRT -]}$$

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}, \Gamma \vdash X <: Y, \Gamma \vdash X <: Y}{\Gamma, x : X, y : Y \vdash (x * y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Y}} \quad \text{[T-OPRT *]}$$

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}, \Gamma \vdash X <: Y}{\Gamma, x : X, y : Y \vdash (x/y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Y}} \quad \text{[T-OPRT /]}$$

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}}{\Gamma, x : X, y : Y \vdash (x \;\texttt{div}\; y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Integer}} \quad \text{[T-OPRT DIV]}$$

Table 23: Type Rules for Multiplicity and Additive Operators

Note that within these type rules the operator + has two overloaded meanings, one for type `Text`, and one for numeric types. We used the notion of bounded universally quantified types to describe the behavior of the operator + for numeric values. The type rule "Oprt-Numeric +" indicates that the + operator accepts types which range over all subtypes of the type `Float`.

Most binary operators cause conversions of operands and yield results the same way. The way these operators cause conversions is called "usual arithmetic conversions". Arithmetic conversions of operands of different native types are performed as shown in the Table 24. As an example in the arithmatic statement $2 + 2.5$ the first argument, 2, will be coerced to its corresponding `Float` type.

| Conditions Met | Conversion |
|---|---|
| Either operand is of type `Float` | Other operand is converted to type `Float` |
| Preceding condition not met and either operand is of type `Decimal` | Other operand is converted to type `Decimal` |
| Preceding conditions not met (none of the operands are of floating types). | Both operands are `Integers` and no conversion needed |

Table 24: Arithmetic Conversions

The multiplicative operators take operands of arithmetic types. The modulus operator (`mod`) has a stricter requirement in that its operands must be of Integer type. The conversions covered in Arithmetic Conversions are applied to the operands, and the result is of the converted type.

The modulus operator yields the remainder given by the following expression, where $e_1$ is the first operand and $e_2$ is the second: $e_1 - (e_1/e_2) * e_2$, where both operands are of integer types. Moreover, division by 0 in either a division or a modulus expression is undefined and causes a run-time error.

$$\frac{\Gamma \vdash S : T, T \in \{\texttt{Float}, \texttt{Decimal}, \texttt{Integer}\}}{S/0 \rightarrow \texttt{wrong}} \quad \text{[E-Division by 0]}$$

$$\frac{\Gamma \vdash S : \mathit{Integer}}{S \ \texttt{mod}\ 0 \rightarrow \texttt{wrong}} \quad \text{[E-Modulus by 0]}$$

### 4.10.2 Assignment Operators

In the Erasmus language an assignment is in the form of v:=e. Note that in the assignment in the form of v:=e, the lvalue $v$ must be a reference type, and rvalue $e$ must be a subtype of the l-value $v$. The type rule for the assignments is given in Table 25.

$$\frac{\Gamma \vdash \texttt{v:ref T}, \Gamma \vdash \texttt{e:T'}, \texttt{T'} <: \texttt{T}}{\Gamma \vdash \texttt{v:=e : Void}} \quad \text{[Stat Assign]}$$

Table 25: Type Rules for Assignment in Erasmus

Moreover, the assignment operators + =, etc., expand to regular assignment statements, as shown below:

$$\texttt{v} \mathrel{+}= \texttt{e} \equiv \texttt{v} := \texttt{v} + \texttt{e}$$

$$\texttt{v} \mathrel{-}= \texttt{e} \equiv \texttt{v} := \texttt{v} - \texttt{e}$$

$$\texttt{v} \mathrel{*}= \texttt{e} \equiv \texttt{v} := \texttt{v} * \texttt{e}$$

$$\texttt{v} \mathrel{/}= \texttt{e} \equiv \texttt{v} := \texttt{v} / \texttt{e}$$

$$\texttt{v} \mathrel{\%}= \texttt{e} \equiv \texttt{v} := \texttt{v} \% \texttt{e}$$

The type rules for these assignment operators are given in Table 26 below.

$$\frac{\Gamma \vdash \texttt{v:ref T}, \Gamma \vdash \texttt{e:T'}, \texttt{T'} <: \texttt{T}}{\Gamma \vdash \texttt{v+=e} \ : \ \texttt{Void}} \quad \text{[STAT ASSIGN 1]}$$

$$\frac{\Gamma \vdash \texttt{v:ref T}, \Gamma \vdash \texttt{e:T'}, \texttt{T'} <: \texttt{T}}{\Gamma \vdash \texttt{v-=e} \ : \ \texttt{Void}} \quad \text{[STAT ASSIGN 2]}$$

$$\frac{\Gamma \vdash \texttt{v:ref T}, \Gamma \vdash \texttt{e:T'}, \texttt{T'} <: \texttt{T}}{\Gamma \vdash \texttt{v*=e} \ : \ \texttt{Void}} \quad \text{[STAT ASSIGN 3]}$$

$$\frac{\Gamma \vdash \texttt{v:ref T}, \Gamma \vdash \texttt{e:T'}, \texttt{T'} <: \texttt{T}}{\Gamma \vdash \texttt{v/=e} \ : \ \texttt{Void}} \quad \text{[STAT ASSIGN 4]}$$

$$\frac{\Gamma \vdash \texttt{v:ref T}, \Gamma \vdash \texttt{e:T'}, \texttt{T'} <: \texttt{T}}{\Gamma \vdash \texttt{v \%=e} \ : \ \texttt{Void}} \quad \text{[STAT ASSIGN 5]}$$

Table 26: Type Rules for Assignment in Erasmus

### 4.10.3 Relational and Equality Operators

The appropriate type rules for relational and equality operators are given in Table 27 below.

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}}{\Gamma, x : X, y : Y \vdash (x < y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-NUM} <]$$

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}}{\Gamma, x : X, y : Y \vdash (x > y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-NUM} >]$$

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}}{\Gamma, x : X, y : Y \vdash (x <= y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-NUM} <=]$$

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}}{\Gamma, x : X, y : Y \vdash (x >= y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-NUM} >=]$$

$$\frac{\Gamma \vdash X <: \texttt{Float}, \Gamma \vdash Y <: \texttt{Float}}{\Gamma, x : X, y : Y \vdash (x! = y) : \forall X <: \texttt{Float}.\forall Y <: \texttt{Float}.X \times Y \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-NUM} !=]$$

$$\frac{\Gamma, x : \texttt{Text} \vdash x : \texttt{Text} \;\; \Gamma, y : \texttt{Text} \vdash y : \texttt{Text}}{\Gamma \vdash (x < y) : \texttt{Text} \times \texttt{Text} \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-TEXT} <]$$

$$\frac{\Gamma, x : \texttt{Text} \vdash x : \texttt{Text} \;\; \Gamma, y : \texttt{Text} \vdash y : \texttt{Text}}{\Gamma \vdash (x > y) : \texttt{Text} \times \texttt{Text} \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-TEXT} >]$$

$$\frac{\Gamma, x : \texttt{Text} \vdash x : \texttt{Text} \;\; \Gamma, y : \texttt{Text} \vdash y : \texttt{Text}}{\Gamma \vdash (x <= y) : \texttt{Text} \times \texttt{Text} \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-TEXT} <=]$$

$$\frac{\Gamma, x : \texttt{Text} \vdash x : \texttt{Text} \;\; \Gamma, y : \texttt{Text} \vdash y : \texttt{Text}}{\Gamma \vdash (x >= y) : \texttt{Text} \times \texttt{Text} \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-TEXT} >=]$$

$$\frac{\Gamma, x : \texttt{Text} \vdash x : \texttt{Text} \;\; \Gamma, y : \texttt{Text} \vdash y : \texttt{Text}}{\Gamma \vdash (x! = y) : \texttt{Text} \times \texttt{Text} \rightarrow \texttt{Bool}} \quad [\text{T-OPRT-TEXT} !=]$$

Table 27: Type Rules for Relational and Equality Operators

### 4.10.4 Logical Operators

Logical operators are ''and'', ''or'', and ''not''. The ''not'' operator has the highest precedence and the ''or'' operator has the lowest. As usual, precedence can be overridden by parenthesis. The first order type system is required to construct these operators. The type rules for logical operators are given in Table 28.

$$\frac{\Gamma \vdash B_1, B_2 : \texttt{Bool}}{\Gamma \vdash (B_1 \texttt{ and } B_2) : \texttt{Bool}} \quad \text{[T-OPRT-AND]}$$

$$\frac{\Gamma \vdash B_1, B_2 : \texttt{Bool}}{\Gamma \vdash (B_1 \texttt{ or } B_2) : \texttt{Bool}} \quad \text{[T-OPRT-OR]}$$

$$\frac{\Gamma \vdash B : \texttt{Bool}}{\Gamma \vdash (\texttt{not } B) : \texttt{Bool}} \quad \text{[T-OPRT-NOT]}$$

$$\frac{\Gamma \vdash B_1, B_2 : \texttt{Bool}}{\Gamma \vdash (B_1 \texttt{ = } B_2) : \texttt{Bool}} \quad \text{[T-B-OPRT =]}$$

$$\frac{\Gamma \vdash B_1, B_2 : \texttt{Bool}}{\Gamma \vdash (B_1 \texttt{ != } B_2) : \texttt{Bool}} \quad \text{[T-B-OPRT !=]}$$

Table 28: Type Rules For Logical Operators

## 4.11 Conditional Statement

The syntax of conditional statements are given below.

*Conditional* = if *bool* then *Sequence*

{elif *bool* then *Sequence*}

[else *Sequence* ] end.

The first order type system is required to explain the behavior of the if statement. The type rules for conditional statements are given in Table 29.

$$\frac{\Gamma \vdash B : \texttt{Bool}, \Gamma \vdash S : \texttt{Void}}{\Gamma \vdash (\texttt{if } B \texttt{ then } S \texttt{ end}) : \texttt{Void}} \quad [\textsc{Stat If}]$$

$$\frac{\Gamma \vdash B_1, B_2 : \texttt{Bool}, \Gamma \vdash S_1, S_2 : \texttt{Void}}{\Gamma \vdash (\texttt{if } B_1 \texttt{ then } S_1 \texttt{ elif } B_2 \texttt{ then } S_2 \texttt{ end}) : \text{Void}} \quad [\textsc{Stat ElIf}]$$

$$\frac{\Gamma \vdash B_1, B_2 : \texttt{Bool}, \Gamma \vdash S_1, S_2, S_3 : \texttt{Void}}{\Gamma \vdash (\texttt{if } B_1 \texttt{ then } S_1 \texttt{ elif } B_2 \texttt{ then } S_2 \texttt{ else } S_3 \texttt{ end}) : \texttt{Void}} \quad [\textsc{Stat Else}]$$

Table 29: Type Rules for Conditional Statement

The rules "Stat IF", "Stat ElIF", and "Stat Else" indicate that these conditional statements are well formed terms in our static enviroment $\Gamma$.

## 4.12 Loop Statement

Erasmus language provides one kind of loop statements. This statement is in the form of:

| | | |
|---|---|---|
| Loop | = | loop *Sequence* end. |
| t::= | | terms |
| | loop t end | loop statement |
| | exit | keyword exit |

Table 30: Syntax For Loop Statement

The sequence in a loop statement is executed repeatedly until one of its statements executes an *exit* statement. Note that the *exit* statement is allowed only within a loop or loopselect statement. Moreover, Erasmus language provides the usage of *while* statement only within the declaration of loop statement. The type rlues for these loop statements are shown in Table 31.

$$\frac{\Gamma \vdash S : \texttt{Void}}{\Gamma \vdash (\texttt{loop } S \texttt{ exit end}) : \texttt{Void}} \quad \text{[T-STAT LOOP-EXIT]}$$

$$\frac{\Gamma \vdash S : \texttt{Void}}{\Gamma \vdash (\texttt{(loop S end) : Void})} \quad \text{[T-STAT LOOP]}$$

Table 31: Type Rules for Loop Statements in Erasmus

## 4.13 While and Until Statements

The *while* and *until* statements are, effectively, macros:

| | | | | |
|---|---|---|---|---|
| While | = | while C | ≡ | if not C then exit end. |
| Until | = | until C | ≡ | if C then exit end. |

| | | |
|---|---|---|
| t::= | | terms |
| | while t | while statement |
| | until t | until statement |

Table 32: Syntax For While and Until Statements

The *while* and *until* statements are only allowed within the loop statement. In the loop statements with *while* or *until*, the body of loop will be executed repeatedly until its *while* or *until* statement executes the *exit* statement. Table 33 shows the type rules for *while* and

*until* statements.

| | |
|---|---|
| $\dfrac{\Gamma \vdash B : \texttt{Bool}, \Gamma \vdash S_1, S_2 : \texttt{Void}}{\Gamma \vdash (\texttt{loop}\ S_1\texttt{;}\ \texttt{until}\ B\texttt{;}\ S_2\ \texttt{end}) : \texttt{Void}}$ | [T-STAT LOOP-UNTIL] |
| $\dfrac{\Gamma \vdash B : \texttt{Bool}, \Gamma \vdash S_1, S_2 : \texttt{Void}}{\Gamma \vdash (\texttt{loop}\ S_1\texttt{;}\ \texttt{while}\ B\texttt{;}\ S_2\ \texttt{end}) : \texttt{Void}}$ | [T-STAT LOOP-WHILE] |

Table 33: Type Rules For While and Until Statements in Erasmus

# Chapter 5

# Erasmus As a Process Oriented Language

Erasmus language is a process oriented programming language which is mainly based on cells and their interactions. In this chapter we will explain the main concepts of the Erasmus project such as messages, protocols, ports, closures, and cells. We will then expand and formalize our previous type system with the syntax, appropriate type rules, and the subtyping relations of these fundamental concepts. We will also provide some algorithms for protocol satisfaction and protocol equality which enables us to compare the protocols and to explain the behaviors of cells.

Object-oriented languages are becoming increasingly popular for the development of software systems of all kinds, ranging from small web-based applications to large server-side applications. The concept of the objects gives a power to software engineers through their desire goal, as an example modern object-oriented languages such as Java contain features such as exception handling, dynamic binding, extensive control of visibility, and threads.

Although these features add to a language's expressive power and provide many benefits from a software engineering point of view, they also make it more difficult to implement a language efficiently. One of the main difficulties of an object oriented languages is that in the object model, an object doesn't provide a full control over the sequence in which method calls and events may hit an object.

Erasmus language is a process oriented language in which all the objects are cells and the behaviors of cells could be explained by their processes. These cells may interact with each other and have the full control over their interactions which contrast with the object model. Therefore in this section we will concentrate on introducing Erasmus language as a process oriented language by finding the answers to the questions such as what do we mean by the notion of *cells*, *processes*, *messages*, *protocols*, and *ports* and within these explanations we will expand our type system with the appropriate syntaxes , type rules, and the subtyping relations.

An execution of the Erasmus project comprises cells and their interactions. A cell could be as small as a single character or as large as a distributed system. Each cell has its own attributes and behaviors, and has a full control over them which contrasts with the object model in which provides week control over the sequences in which methods calls and events may hit an object. The behavior(s) of a cell are fully determined by its *processes*. Generally talking, Erasmus project is based on cells and their communications; these communications take place within the processes of cells by the means of *messages*.

The eventual economic success of the component-based software industry depends on the ability to mix and match parts selected from different suppliers. The components of

Erasmus project are cells. Cells are either *servers* or *clients* or could be *both.* Servers and clients may *link* together and interact with each other by the means of messages. *Protocols* are used to define not only the structure of the messages that may flow between cells but also their allowable sequences. These communications between a server and a client must be done in a way that:

- The server side must accept all the queries sent by the client.
- The client side must accept all the replies sent by the server.

## 5.1 Messages

As mentioned earlier, an execution of Erasmus langauge comprises cells and their interactions, two cells may link together and interact that is; the sequences of information may exchange between them. The unit of these interactions is called *messages.*

When two cells link together, one becomes a server and the other becomes a client. The client side sends queries to the server side, and the server side sends replies back to the client. These queries and replies are called *messages.* A message may or may not contain data. A message which doesn't contain data is called a *signal.*

Messages are defined by their names and the types of data they are carrying; they also can carry the data of different types. Messages are defined by $m : T, U, \ldots$ where $m$ is its name and $T, U, \ldots$ are the types of the data it carries, and signals are defined by just their names. In the Erasmus project the declarations for messages are only allowed just within protocols.

According to the materials mentioned above and by taking the direction of the communications between cells into our consideration, it is useful in this place to distinguish between four kinds of messages flowing between cells.

Therefore, we categorize messages into four groups which are given below:

1. A query signal $s$ sent from a client to a server.
2. A reply signal $\hat{}s$ sent from a server to a client.
3. A query message $m : T$ sent from a client to a server.
4. A reply message $\hat{}m : T$ sent from a server to a client.

Table 34 shows the typing rules for the messages. Note that the type rule "Type Message" defines $m$ as a send message that is; the direction of this message during the communication between two cells is from a client to the server. Moreover, the type rule "Type Reply Message" defines $\hat{}m$ as a reply message indicating that the direction of this message during the communications is from a server to the client side from which made a request.

$$\frac{\Gamma \vdash A}{\Gamma, m : A \vdash m : A} \quad \text{[T-Send Message]}$$

$$\frac{\Gamma \vdash A}{\Gamma, \hat{}m : A \vdash \hat{}m : A} \quad \text{[T-Reply Message]}$$

Table 34: Type Rules for Erasmus

Note that the definitions of messages are same as the definition of variables, but the definition of messages are only allowed within the scope of protocols.

### 5.1.1 Subtyping for Messages

Figure 3 shows a communication between a server and a client.

| Client | | Server |
|---|---|---|
| $m_c : T$ | $\longrightarrow$ | $m_s : T'$ |
| $\hat{}n_c : U'$ | $\longleftarrow$ | $\hat{}n_s : U$ |

Figure 3: Client/server communication

In this figure the client side sends a query message $m_c$ of type $T$ to the server. The server side expects a query message $m_s$ of type $T'$ from a client. Therefore, in order for this communication to take place first the message identifier $m_c$ must be equal to the message identifier $m_s$ that is $m_c = m_s$, and second the type of the message $m_c$ must be a subtype of the type of the message $m_s$ that is, $T <: T'$. On the other side, when this server recieves a message(s) from a client, it sends a reply message $\hat{}n_s$ of type $U$ back to the client side. Same as above, the client side expects a replay message $\hat{}n_c$ of type $U'$. Thus, in order for this communication to take place $n_c = n_s$, and $U$ must be a subtype of $U'$, $U <: U'$.

Sending messages from a client to a server, and from a server to a client work exactly like functions, Thus the use of *covariant* and *contravariant* is obvious. Formally, the subtyping rule for queries and replies are given by the following inference rules:

$$\frac{m_c = m_s \qquad T <: T'}{(m_c : T) <: (m_s : T')} \quad \text{[SUB-QUERY]}$$

$$\frac{m_c = m_s \qquad T <: T'}{(\hat{}m_s : T') <: (\hat{}m_c : T)} \quad \text{[SUB-REPLY]}$$

Figure 4: Subtyping Rules For Messages

According to these inference rules, we say that a message $m_c$ of type $T_c$ defined in the client cell, is a subtype of the message $m_s$ of type $T_s$ defined in the server cell, if and only if $m_c = m_s$ and $T_c <: T_s$. Similarly we say that a reply message $\hat{}m'_s$ of type $T'_s$ defined in the server cell is a subtype of a reply message $\hat{}m'_c$ of type $T'_c$ defined in the client cell if and only if $m'_s = m'_c$ and $T'_s <: T'_c$.

## 5.2 Protocols

To summarize so far, we mentioned that Erasmus language is a process oriented language which is actually based on cells and their interactions. These interactions are called messages and they may flow between two connected cells. Messages may carry data from a cell to another and the direction in which these data are sent indicates whether a cell is a client or a server side. Typically a send message from a client to a server is shown somehow similar to the variable declaration, $m : T$, in which $m$ stands for the message identifier and $T$ stands for the type of data it carries, and a reply message from a sever to a client is shown with the circumflex symbol in front of the variable declaration, that is $\hat{}m : T$. Note that message definitions are only allowed within the definition of protocols.

In this section we will introduce a new concept called *protocol* and we will expand our previous type system with the appropriate type and subtyping rules.

**Definition** *protocol* is an expression which specifies both the structure of messages and also their allowable sequences. For any messages and signals, there is a corresponding protocol which specifies the single allowable sequence, just the message itself, send exactly once. Therefore, we use the symbol $[m]$ for protocols which contains the single allowable sequence.

In Erasmus project, protocols are defined by their names and their allowable sequences, consist of sequences of messages which the protocols allows. Message names must be unique within the definition of a protocol. The syntax of protocols are shown in Table 35. Following is an example of a protocol of a cell in the Erasmus language.

$$\text{prot} \quad = \quad [\,\text{inp} : \texttt{Integer}\ ,\ \hat{}\text{inp} : \texttt{Integer}\,]$$

In this example *prot* is a protocol of a cell(s) which is both a client and a server. This protocol allows a cell(s) to send (or receive) a message *inp* of type `Integer` followed by a reply message $\hat{}$*inp* of type `Integer`.

Protocols are either single allowable sequences defined above, or could be constructed inductively by using the operators( ";" , "|" , "?" , "*", "+"). Protocols which are constructed inductively by using these operators are called *composite protocols.* Therefore, a protocol expression may be preceded by *Multiplicity* that indicates how often it may be sent.

| | | |
|---|---|---|
| *ProtocolDefinition* | = | ProtocolName "=" *Protocol* |
| *Protocol* | = | ProtocolName \| "[" *ProtocolExpression* "]". |
| *ProtocolExpression* | = | ['∧'] *VariableDeclaration* |
| | \| | [*Multiplicity*] *ProtocolExpression* |
| | \| | {*ProtocolExpression*}; |
| | \| | {*ProtocolExpression*} \| |
| | \| | '('*ProtocolExpression*')'. |
| *Multiplicity* | = | '?' \| '*' \| '+'. |

| | | |
|---|---|---|
| t::= | | terms |
| | $[\alpha_1 = t_1, \ldots, \alpha_n = t_n]$ | protocol |
| | $t.\alpha_i$ | protocol projection |
| | ?t | multiplicity |
| | *t | multiplicity |
| | +t | multiplicity |
| | $t_1; t_2$ | composit protocol with ; |
| | $t_1 \mid t_2$ | composit protocol with \| |
| T::= | | type |
| | $[\alpha_1 : T_1, \ldots, \alpha_n : T_n]$ | type protocol |

Table 35: Syntax of Protocols in Erasmus

**Protocol Operators** As mentioned above, other protocols are constructed inductively using operators. In the following consider $p_i$ and $q_i$, $i = 1, 2, \ldots$, as protocols.

- $?p$ is a protocol specifying that $p$ occurs once or not at all.
- $*p$ is a protocol specifying that $p$ occurs zero or more times.
- $+p$ is a protocol specifying that $p$ occurs one or more times.
- $p \mid q$ is a protocol specifying that either $p$ or $q$ occurs once.
- $p \ ; \ q$ is a protocol specifying the sequence consisting of $p$ followed by $q$.

Note that the operator ";" has higher precedence than "|", and the multiplicity operator has higher precedence, followed by sequencing. Parentheses may be used to override (or confirm) precedence. For example, $*(p_1; p_2 \mid q_1; q_2)$ is equivalent to $*((p_1; p_2) \mid (q_1; q_2))$ but is not equivalent to $*(p_1; (p_2 \mid q_1); q_2)$.

The operator "|" is commutative and associative. Protocols such as $p_1 \mid p_2 \mid \cdots \mid p_n$ are unambiguous and invariant under permutation of the $p_i$. Also, $p \mid p = p$.

The operator ";" is associative. Protocols such as $p_1 \ , p_2 \ , \cdots , p_n$ are unambiguous.

**Trivial Protocol** Although the protocols defined above are sufficient for programming languages, but from a mathematical point of view it is useful to define some protocols which will be never used, but the concept is useful in theory.

- The *empty protocol*, written $\phi$, corresponds to all the unconnected cells.
- The *null protocol*, written $\epsilon$, corresponds to the communication between a pair of connected cells in which only empty sequence is allowed. The definition of protocol

null in Erasmus is : $null = [\,]$;

Protocols are different from types; they have an extra, temporal, dimension compared with the types. The type of protocols could be described by record types, but there are some operators defined on protocols which records don't accept, also the subtyping rule for protocols are different from records; therefore we are justified at this moment to define a new type called protocol type, and augment its type rules to our previous type sysmtem. Similar to the record type this new type consists of a sequence of mappings from labels to values called messages. The symbol "[ ]" is used in our type system to indicate a protocol type that makes it different from the symbol of record types which is "{}".

The type rules for constructing protocols are given in Table 36 below. As usual the first two type rules indicate that the empty protocol and the null protocol are well formed terms in our static environment $\Gamma$. The type rule "T-Protocol" construct a new type called protocol type. Moreover, the type rules for protocol operations and multiplicity indicate that composite protocols made by these operators are well formed terms and have the type protocol in our static environment $\Gamma$.

In the Erasmus language two cells can only and only link together and communicate if and only if the protocol of the server side *satisfies* the protocol of the client side. The intution is that, a server side must accept all the queries sent by a client and the client side must accept all the replies sent back by the server. These conditions hold true if the protocol of the server side satisfies the protocol of the client side. Note that this relation is not symmetric therefore we call it *satisfaction* instead of compatibility. Thus, "$p_s$ satisfies $p_c$" written $p_c <: p_s$, means that, in some sense, $p_c$ is "smaller" than $p_s$. Therefore, similar

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \phi} \quad \text{[T-Empty Protocol]}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \epsilon : [\,]} \quad \text{[T-Null Protocol]}$$

$$\frac{\Gamma \vdash \alpha_i : A_i, \Gamma \vdash \beta_i : T_i}{\Gamma \vdash [\alpha_1 \mapsto \beta_1, \ldots \alpha_n \mapsto \beta_n] : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]} \quad \text{[T-Protocol]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]}{\Gamma \vdash p.\alpha_i : \texttt{ref}\ T_i} \quad \text{[T-Prot-Project]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n], q : [\alpha'_1 : T'_1, \ldots, \alpha'_n : T'_n]}{\Gamma \vdash (p; q)} \quad \text{[T-Multi-;]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n], q : [\alpha'_1 : T'_1, \ldots, \alpha'_n : T'_n]}{\Gamma \vdash (p \mid q)} \quad \text{[T-Multi-|]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]}{\Gamma \vdash ?p} \quad \text{[T-Multi-?]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]}{\Gamma \vdash *p} \quad \text{[T-Multi-*]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]}{\Gamma \vdash +p} \quad \text{[T-Multi-+]}$$

Table 36: Type Rules for Protocols in Erasmus

to the other types in our type system we should be able to define a subtyping relation, *satisfaction*, for protocols.

As mentioned eralier, Erasmus language enforces type equality for all types instead of protocols by the means of name equivalences that is two types are equal if and only if they have a same names. As an example two record types are equal if they have equal names, there is also a subtyping relation between records if and only if one of them is a parent and the other is a child. But on the other side, Erasmus project compares protocols by their structures(their traces sets) meaning that two protocols are equal if they have the same structures(same traces sets).

In order to define the satisfaction relation between protocols, we need to define some important concepts which are given in the following sub-sections.

### 5.2.1 Traces and Traces Sets

Traces are a sequence of *events* happened up to the moment of time according to a process, [9]. These processes are communications between connected cells, therefore traces in Erasmus project are defined as the sequence of messages, that may flow between a pair of cells during their connections.

The symbol for traces is <>. Thus, the trace consisting of messages $m_1$,$m_2$,$m_3$,... is written $< m_1, m_2, m_3, ... >$.

If a trace $t$ belongs to the sequence of messages allowed by a protocol $p$ then we say that $t$ respects $p$. Figure 5 shows, for each protocol operator applied to messages, the traces

that respect the protocol. To be easier to read we omit the types of the messages in the following figure.

| Protocol | Traces |
|---|---|
| $[m]$ | $\langle m \rangle$ |
| $?[m]$ | $\langle\,\rangle$, $\langle m \rangle$ |
| $[m_1] \mid [m_2] = [m_1 \mid m_2]$ | $\langle m_1 \rangle$, $\langle m_2 \rangle$ |
| $[m_1]; [m_2] = [m_1; m_2]$ | $\langle m_1, m_2 \rangle$ |
| *[m] | $\langle\,\rangle$, $\langle m \rangle$, $\langle m, m \rangle$, $\langle m, m, m \rangle$, ... |
| +[m] | $\langle m \rangle$, $\langle m, m \rangle$, $\langle m, m, m \rangle$, ... |

Figure 5: Protocols and traces

There are some operations defined on traces which are given below:

- **Concatenation** Traces may be concatenated. The concatenation of trace $t$ and $s$ is written $t.s$. For example, if $t =< m_1, m_2 >$ and $s =< m_3, m_4 >$ then $t.s =< m_1, m_2, m_3, m_4 >$.

- **Power** Traces may be powered. The power of trace $t$ by $n$, read $t$ ($n$ times), is written $t^n$. For example, if $t =< m_1, m_2 >$ then $t^2 = t.t =< m_1, m_2, m_1, m_2 >$ and $t^n =< m_1, m_2, m_1, m_2, ... >$.

**Definition** $\mathcal{T}(p)$ is the set of all traces that respects a protocol $p$.

For example, if $m$ is a message, then:

$$\mathcal{T}([m]) = \{\langle m \rangle\},$$

$$\mathcal{T}(?[m]) = \{\langle\,\rangle, \langle m \rangle\}.$$

Like traces we are able to define concatenation and power on $\mathcal{T}()$. The concatenation operator is extended for trace sets $S$ and $T$ as follows:

$$S.T \;=\; \{\, s.t \mid s \in S \text{ and } t \in T \,\}.$$

Trace sets may also be powered, that is:

$$\begin{aligned} T^0 &= \{\langle\,\rangle\} \\ T^n &= \left\{ \underbrace{T.T \ldots T}_{\text{n times}} \right\}. \end{aligned}$$

Since traces sets are sets, we use conventional set-theoretic notation ($\in$, $\subseteq$, $\cup$, etc.) to write expressions involving traces and trace sets. Using trace sets, we can provide formal definitions for the protocol operators introduced before, as shown in Table 37 below. As usual, $m$ is a message and $p, q$ are protocols, and T and S are traces sets.

| | |
|---|---|
| $\mathcal{T}(\phi) = \{\}$ | $\mathcal{T}([m]) = \{\langle m \rangle\}$ |
| $\mathcal{T}(\epsilon) = \{\langle\,\rangle\}$ | $\mathcal{T}(?p) = \mathcal{T}(p) \cup \{\langle\,\rangle\}$ |
| $\mathcal{T}(p)^0 = \{\langle\,\rangle\}$ | $\mathcal{T}(p \mid q) = \mathcal{T}(p) \cup \mathcal{T}(q)$ |
| $\mathcal{T}(p)^n = \left\{ \underbrace{\mathcal{T}(p) \ldots \mathcal{T}(p)}_{\text{n times}} \right\}$ | $\mathcal{T}(p;q) = \mathcal{T}(p).\mathcal{T}(q)$ |
| $\text{S.T} = \{\, s.t \mid s \in S \text{ and } t \in T \,\}$ | $\mathcal{T}(*p) = \bigcup_{n \geq 0} \mathcal{T}(p)^n$ |

Table 37: Defining Protocols by Trace Sets

### 5.2.2 Protocol Equality

As mentioned before, Erasmus project uses name equivalences for type comparisons but when it comes to protocols it uses structure equivalences instead. To understand why protocols comparison is based on the structural equivalences it is useful in this place to distinguish between two different equalities called intension and extension equality.

**Intension and Extension** The *intension* of an expression is its syntactic form; its *extension* is its value in some appropriate mathematical space. In mathematics, equality is usually defined between extensions rather than intensions.

For example, consider the expressions $3 \times (4 + 5)$ and $3 \times 4 + 3 \times 5$. Normally, we would say they are equal, because both expressions yield 27 when evaluated; this is *extensional comparison.* If we were concerned with the time required to calculate them, however, we would consider them unequal, because the first requires two operations but the second requires three operations; this is *intensional comparison.*

The distinction is important when computation is involved. Consider sets $S_1$ and $S_2$ defined by:

$$\begin{aligned} S_1 &= \{\, n \mid n \text{ is prime} \,\} \cap \{\, n \mid n < 20 \,\} \\ S_2 &= \{2, 3, 5, 7, 11, 13, 17, 19\}. \end{aligned}$$

These sets are equal but, in order to prove this, we must reason with their intensions. We cannot reason with their extensions, because the first component of $S_1$ is an infinite set.

With this in mind, we define equality of protocols (intensions) in terms of equality of their traces (extensions). Therefore, for any given protocols $p$ and $q$, $p = q$ if and only if:

1. $\mathcal{T}(p) = \mathcal{T}(q)$

2. The subtyping rules for any send messages and replies in Figure 4 must hold true, that is:

   - For any send message $m$ of type $T$ in $p$, there must be a send message in $q$ with the same message identifier and the same type.
   - For any reply query $\hat{}m$ of type $T$ in $p$, there must be a replay message in $q$ with the same identifier and the same type.

Note that traces may have unbounded size, therefore following are some definitions with which we are able to compare protocols by comparing their traces sets even if their traces sets have infinite number of elements.

**Definition** Protocol $p$ is an *infinite protocol* if it's traces set has infinite number of allowable sequences, that is, $p$ is infinite if and only if $|\mathcal{T}(p)|= \infty$.

Acoording to the definition of *infinite protocols* following rules are trivial.

- $*p$ and $+p$ are infinite protocols becasue $|\mathcal{T}(*p)| = \infty$ and $|\mathcal{T}(+p)| = \infty$.
- If protocol $p$ is an infinite protocol then for any protocols $q$, $z$ and ... the composite protocol $p' = p; q; z; \ldots$ is an infinite protocol.
- If protocol $p$ is an infinite protocol then for any protocols $q$, $z$ and ... the composite protocol $p' = p \mid q \mid z \mid \ldots$ is an infinite protocol.
- No other protocol is an infinite protocol.

**Definition** Protocol $p$ is said to be *nullable* if $\langle\,\rangle \in \mathcal{T}(p)$.

Acoording to the definition of *nullable protocols* following rules are trivial.

- Protocol $\epsilon$ is *nullable* because $\mathcal{T}(\epsilon) = \{\langle\ \rangle\}$.
- For any protocol $p_i$, $i \geq 0$, The composite protocol $p_1 \mid p_n \mid \epsilon \mid \ldots$ is *nullable.*
- If $p$ is a protocol then $?p$ is *nullable* because $?p = p \mid \epsilon$.
- If $p$ is a protocol then $*p$ is *nullable* because $*p = +p \mid \epsilon$.
- If protocols $p_i$, $i \geq 0$ are *nullable* then $p_1; p_2; \ldots; p_n$ is *nullable.*
- No other protocol is *nullable.*

Note that it is possible for a protocol to be both *infinite* and *nullable.*

**Definition** Consider $p$ as a protocol, then $N_{max}(p)$ is equal to the maximum number of messages that $p$ can send or receive during communication periods, accordingly $N_{min}(p)$ is the minimum number of messages that $p$ can send or receive.

For example:

$$p = \left[\underbrace{m : T; m' : T'}_{Max} \mid m'' : T''\right] \Rightarrow N_{max}(p) = 2,\ N_{min}(p) = 1.$$

Figure 6 shows the maximum and minimum number of messages for the given protocols, and the ways to compute this number for any composite protocol. As usual, $m$ is a message and $p, q$ are protocols. Note that by using these rules we are able to compute the $N$, maximum and minimum number of messages, for any given compisite protocol.

As mentioned before, Erasmus language compares two protocols by first comparing their traces sets, and second by comparing the types of their messages and reply queries defined in

| | | | | | |
|---|---|---|---|---|---|
| $N_{max}(\phi)$ | = | 0 | $N_{min}(\phi)$ | = | 0 |
| $N_{max}(\epsilon)$ | = | 1 | $N_{min}(\epsilon)$ | = | 1 |
| $N_{max}([m])$ | = | 1 | $N_{min}([m])$ | = | 1 |
| $N_{max}(p \mid q)$ | = | $max\{N_{max}(p), N_{max}(q)\}$ | $N_{min}(p \mid q)$ | = | $min\{N_{min}(p), N_{min}(q)\}$ |
| $N_{max}(p; q)$ | = | $N_{max}(p) + N_{max}(q)$ | $N_{min}(p; q)$ | = | $N_{min}(p) + N_{min}(q)$ |
| $N_{max}(?p)$ | = | $N_{max}(p)$ | $N_{min}(?p)$ | = | 1 |
| $N_{max}(*p)$ | = | $\infty$ | $N_{min}(*p)$ | = | 1 |
| $N_{max}(+p)$ | = | $\infty$ | $N_{min}(+p)$ | = | $N_{min}(p)$ |

Figure 6: Maximum and Minimum number of messages

their scopes. But traces sets may have unbounded size which makes it difficult to compare them. Therefore, we are justified in this place to introduce a new concept called *finite reduction*, and the goal of this concept is to reduce the size of protocols traces sets to make it possible for comparision.

**Definition** For any protocol $p$ there is a corresponding protocol $\overline{p}$ called *finite reduction* of $p$ in which protocol $\overline{p}$ contains all the allowable sequences defined in the protocol $p$ except the multiplicity operator $+$.

Following is an example of *finite reduction* of protocols $p$ and $q$.

$$p = +([m_1 : T_1; m_2 : T_2]) \Rightarrow \overline{p} = [m_1 : T_1; m_2 : T_2]$$

$$q = *([m'_1 : T'_1; m'_2 : T'_2]) \Rightarrow \overline{q} = [m'_1 : T'_1; m'_2 : T'_2] \mid \epsilon$$

According to the definition of $\overline{p}$ following terms are trivial.

- $\overline{\epsilon} = \epsilon$.

- $\overline{[m]} = [m]$.
- $\overline{(p_1; p_2)} = \overline{p_1}; \overline{p_2}$.
- $\overline{(p_1 \mid p_2)} = \overline{p_1} \mid \overline{p_2}$.
- $\overline{+p} = \overline{p}$.
- $\overline{*p} = \overline{+p \mid \epsilon} = \overline{p} \mid \epsilon$.
- $\overline{?p} = \overline{p \mid \epsilon} = \overline{p} \mid \epsilon$.

The last three rules indicate that if there is a multiplicity symbol in the definition of protocol $p$, then protocol $\overline{p}$ is equal to the definition of $p$ without the multiplicity symbol. Within these rules given above, one is able to construct the *finite reduction* of any given protocol.

To summerize so far, two protocols are said to be equal if they have equal structures, and by defining protocols with their traces sets we are able then to compare these sets in order to find out their equality, but the traces sets may have unbounded sizes which makes it difficult to compare them, therefore we introduced a new but important concept called "finite reduction" which gives us a strong tool for comparing protocols. The usage of this concept is to reduce the size of traces sets of infinite protocols.

**Theorem 1** *(protocol equality) There is an algorithm that, given two protocols $p$ and $q$, determines whether $p = q$.*

*Proof:* Two protocols are equal if their traces sets are equal, and the subtyping rules for their send and reply messages hold true. Therefore, instead of comparing their allowable

sequences we can compare their traces sets, that is p = q if and only if : (∀ below means for all)

1. $\mathcal{T}(p) = \mathcal{T}(q)$

2. ∀ m:T ∈ p ⇒ m:T ∈ q

3. ∀ ^m:T' ∈ p ⇒ ^m:T' ∈ q

Algorithm $A_1$ given below is the algorithm in which given two protocols $p$ and $q$ returns true if $p = q$ or false otherwise.

**Algorithm $A_1$**

1. If for any two protocols $p$ and $q$ we have $|\mathcal{T}(p)| \neq \infty \neq |\mathcal{T}(q)|$, then these two protocols are equal if the following statements hold true.

   (a) $\mathcal{T}(p) \subseteq \mathcal{T}(q)$ and $\mathcal{T}(q) \subseteq \mathcal{T}(p)$.

   (b) ∀ m:T ∈ p ⇒ m:T ∈ q.

   (c) ∀ ^m:T' ∈ p ⇒ ^m:T' ∈ q.

2. if $|\mathcal{T}(p)| = \infty$ and $|\mathcal{T}(q)| \neq \infty$ or vice versa then they are not equal.

3. If both of protocols $p$ and $p'$ are infinite protocols then go through these steps:

   (a) Find $N_{max}(\overline{p})$ and $N_{max}(\overline{q})$.

   (b) Create the sets $\Psi = \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$ and $\Psi' = \bigcup_{n=1}^{N_{max}(\overline{p})} \mathcal{T}(\overline{q})^n$.

   (c) If $\mathcal{T}(\overline{q}) \subseteq \Psi$ and $\mathcal{T}(\overline{p}) \subseteq \Psi'$ then $p = q$ if:

- $\forall$ m:T $\in$ p $\Rightarrow$ m:T $\in$ q.
- $\forall$ ^m:T' $\in$ p $\Rightarrow$ ^m:T' $\in$ q.
- For any signal message $s \in p$, there is a signal message $s \in q$.

■

This algorithm is both *sound* and *complete* that is:

- This algorithm is *sound* if given any protocols $p$ and $q$ returns true if $p = q$ and false otherwise.
- This algorithm is *complete* if given any protocols $p, q$ such that $p = q$, returns true.

**Proof of soundness** Steps (1) and (2) are the direct conclusion of the set theoretical concepts, that is, two sets are equal if and only if all the members of one are also the members of the other.

To prove the soundness of step (3) we should prove that:

(1) if $\mathcal{T}(\overline{q}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$ then $\mathcal{T}(q) \subseteq \mathcal{T}(p)$

(2) if $\mathcal{T}(\overline{p}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{p})} \mathcal{T}(\overline{q})^n$ then $\mathcal{T}(p) \subseteq \mathcal{T}(q)$

(3) if (1) and (2) hold true then $\mathcal{T}(p) = \mathcal{T}(q)$.

To prove this we use the following lemmas.

**Lemma 1** If $p$ is a protocol then $\mathcal{T}(\overline{p}) \subseteq \mathcal{T}(p)$.

*Proof* If $p$ is not an infinite protocol then by the definition of $\overline{p}$ we have $p = \overline{p}$ which implies that $\mathcal{T}(p) = \mathcal{T}(\overline{p})$. Now consider $p$ is an infinite protocol, thus it is obvious that $p$ has the multiplicity operator "+" in its definition, and there is a protocol $q$ that $p = +q$ where $q$ is not an infinite protocol. Therefore,

$$\mathcal{T}(p) = \mathcal{T}(+q) = \underbrace{\mathcal{T}(q).....\mathcal{T}(q)}_{n \text{ times}, n \geq 0} \quad (1)$$

$$\mathcal{T}(\overline{p}) = \mathcal{T}(\overline{+q}) = \mathcal{T}(q) \quad (2)$$

Thus, (1) and (2) imply that $\mathcal{T}(\overline{p}) \subseteq \mathcal{T}(p)$. ■

**Lemma 2** If $p$ is an infinite protocol then for any set $A \subseteq \mathcal{T}(p)$, $A.A \subseteq \mathcal{T}(p)$ where dot between two $A$'s is concatenation.

*Proof:* protocol $p$ is an infinite protocol and is in the form of $p = +\overline{p}$. Therefore, the traces set of protocol $p$ is the $n$ times concatenation of $\mathcal{T}(\overline{p})$. Thus, for any element $\langle x \rangle \in \mathcal{T}(p)$ we have $\langle x, x \rangle \in \mathcal{T}(p)$ which lead us to the proof in which for every set $A$ such that $A \subseteq \mathcal{T}(p)$ we have $A.A \subseteq \mathcal{T}(p)$. ■

**Lemma 3** If $p$ is an infinite protocol then $\lim_{n\to\infty}\mathcal{T}(\overline{p})^n = \mathcal{T}(p)$.

*Proof:* $p$ is an infinite protocol, therefore it is in the from of $p = +\overline{p}$. Therefore:

$$\mathcal{T}(p) = \mathcal{T}(+\overline{p}) = \mathcal{T}(\underbrace{\overline{p};\overline{p};\ldots;\overline{p}}_{\text{n times}, n \geq 0})$$

$$= \underbrace{\mathcal{T}(\overline{p}).....\mathcal{T}(\overline{p})}_{\text{n times}, n \geq 0} = \mathcal{T}(\overline{p})^{n \geq 0}$$

$$= \lim_{n\to\infty}\mathcal{T}(\overline{p})^n$$

■

Therefore, by using these three lemmas we are able to prove that the third rule works in the way as expected, that is; for any two given infinite protocol $p$ and $q$ if $\mathcal{T}(\overline{q}) \subseteq$

$\bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$ then:

$$\begin{aligned} \text{lemma 1,3} \quad &\Rightarrow \quad \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n \subseteq \mathcal{T}(p) \\ &\Rightarrow \quad \mathcal{T}(\overline{q}) \subseteq \mathcal{T}(p) \\ \text{lemma 2} \quad &\Rightarrow \quad \lim_{n\to\infty} \mathcal{T}(\overline{q})^n \subseteq \mathcal{T}(p) \\ &\Rightarrow \quad \mathcal{T}(q) \subseteq \mathcal{T}(p) \end{aligned}$$

Similarly if $\mathcal{T}(\overline{p}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{p})} \mathcal{T}(\overline{q})^n$ then $\mathcal{T}(p) \subseteq \mathcal{T}(q)$ which implies that $\mathcal{T}(p) = \mathcal{T}(q)$. Thus, if the subtyping rule for messages and replies holds true, then we have $p = q$. ■

**Proof of completeness** The proof for steps (1) and (2) are obvious. To prove that this algorithm is *complete* for the third step, we have to prove that if $p = q$ then $\mathcal{T}(\overline{q}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$ and $\mathcal{T}(\overline{p}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{p})} \mathcal{T}(\overline{q})^n$.

If $\mathcal{T}(\overline{q}) \not\subseteq \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$ , then we can prove that $\mathcal{T}(\overline{q}) \not\subseteq \mathcal{T}(p) = \mathcal{T}(q)$ which is a contradiction. Therefore, consider that $\mathcal{T}(\overline{q}) \not\subseteq \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$ and $\mathcal{T}(\overline{q}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{q})+1} \mathcal{T}(\overline{p})^n$ then:

$$\begin{aligned} &(1) \quad \mathcal{T}(\overline{q}) \not\subseteq \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n \quad \Rightarrow \quad \exists t \in \mathcal{T}(\overline{q}) \text{ st. } t \notin \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n \\ &(2) \quad \mathcal{T}(\overline{q}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{q})+1} \mathcal{T}(\overline{p})^n \quad \Rightarrow \quad \mathcal{T}(\overline{q}) \subseteq \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n \bigcup \mathcal{T}(\overline{p})^{N_{max}(\overline{q})+1} \\ &(3) \quad \overset{from(1),(2)}{\Rightarrow} \quad t \in \mathcal{T}(\overline{p})^{N_{max}(\overline{q})+1} \end{aligned}$$

trace $t$ belongs to both $\mathcal{T}(\overline{q})$ and $\mathcal{T}(\overline{p})^{N_{max}(\overline{q})+1}$; therefore following statements are true:

$$t \in \mathcal{T}(\overline{q}) \quad \Rightarrow \quad N_{min}(\overline{q}) \leq |t| \leq N_{max}(\overline{q}) \qquad (1)$$

$$t \in \mathcal{T}(\overline{p})^{N_{max}(\overline{q})+1} \quad \Rightarrow \quad [N_{max}(\overline{q}) + 1] \times N_{min}(\overline{p}) \leq |t| \leq [N_{max}(\overline{q}) + 1] \times N_{max}(\overline{p}) \qquad (2)$$

From (1) and (2) we have $[N_{max}(\overline{q}) + 1] \times N_{min}(\overline{p}) \leq |t| \leq N_{max}(\overline{q})$ which is a contradiction, and implies that $\mathcal{T}(\overline{q}) \not\subseteq \bigcup_{n=1}^{N_{max}(\overline{q})+1} \mathcal{T}(\overline{p})^n$.

Accordingly, $\mathcal{T}(\overline{q}) \not\subseteq \bigcup_{n=1}^{\infty} \mathcal{T}(\overline{p})^n = \mathcal{T}(p) = \mathcal{T}(q)$ which is a contradiction. Therefore, if we couldn't produce $\mathcal{T}(\overline{q})$ by $N_{max}(\overline{q})$ times concatenation of $\mathcal{T}(\overline{p})$ then we are not able to produce it at all. Thus, if $p = q$ then $\mathcal{T}(\overline{q})$ must be a subset of $\bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$.

The proof of the other side is the same. ■

### 5.2.3 Protocol Satisfaction

To summerize so far, we mentioned that an execution of Erasmus project comprises cells and their interactions, two cells may link together and communicate. For these cells to communicate, their protocols must be compatible meaning that there must be a relation between their protocols. Therefore like the other types, we should be able to define a subtyping relation for protocols. Since the relation is not symmetric, we call it "satisfaction" rather than compatibility.

The intuition is that; if a client $C$ with protocol $p_c$ is linked to a server $S$ with protocol $p_s$, then $p_s$ *satisfies* $p_c$ if;

1. $S$ responds to all queries sent by $C$, and

2. $C$ accepts all the replies sent by $S$.

To see that satisfacion is different from equality, we note that $S$ may provide services that $C$ never uses. Thus, "$p_s$ satisfies $p_c$", written $p_c <: p_s$, means that, in some sense, $p_c$ is "smaller" than $p_s$.

In the previous section we defined equality of protocols by the equality of their traces sets. Using the same convection, we define protocols *satisfaction* as follows:

**Definition** Protocol $p$ is a subprotocol of $q$, written $p <: q$ if and only if:

- $\mathcal{T}(p) \subseteq \mathcal{T}(q)$.

- $\forall\, m : T \in p$, $\exists m : T' \in q$ such that $T' <: T$.

- $\forall\, \hat{}m : T \in p$, $\exists \hat{}m : T' \in q$ such that $T <: T'$.

**Theorem 2** *The satisfaction relation is partial order.*

*Proof:* A partial order is reflexive, transitive, and antisymmetric. Thus $<:$ is:

- *reflexive*, because

$$\mathcal{T}(p) \subseteq \mathcal{T}(p) \quad \Rightarrow p <: p.$$

- *transitive*, because

$$\begin{aligned} p <: p' \text{ and } p' <: q \quad &\Rightarrow \quad \mathcal{T}(p) \subseteq \mathcal{T}(p') \text{ and } \mathcal{T}(p') \subseteq \mathcal{T}(q) \\ &\Rightarrow \quad \mathcal{T}(p) \subseteq \mathcal{T}(q) \\ &\Rightarrow \quad p <: q. \end{aligned}$$

- *antisymmetric*, because

$$\begin{aligned} p <: p' \text{ and } p' <: p \quad &\Rightarrow \quad \mathcal{T}(p) \subseteq \mathcal{T}(p') \text{ and } \mathcal{T}(p') \subseteq \mathcal{T}(p) \\ &\Rightarrow \quad \mathcal{T}(p) = \mathcal{T}(p') \\ &\Rightarrow \quad p = p'. \end{aligned}$$

■

**Theorem 3** *(*protocol satisfaction*) There is an algorithm, given two protocols $p$ and $q$ decides whether $p <: q$ or $q <: p$, or $p = q$, or there is no relation at all.*

Following is the algorithm that given two protocols $p$ and $q$ decides whether $p <: q$ or $q <: p$, or $p = q$, or there is no relation at all.

**Algorithm $A_2$**

1. If for any given protocols $p$ and $q$, $|\mathcal{T}(p)| \neq \infty \neq |\mathcal{T}(q)|$ then:

   (a) Find the traces sets of both protocols, $\mathcal{T}(p)$ and $\mathcal{T}(q)$.

   (b) $p <: q$ if:

   - $\mathcal{T}(p) \subseteq \mathcal{T}(q)$.
   - $\forall\, m : T \in p$, $\exists m : T' \in q$ such that $T' <: T$.
   - $\forall\, \hat{}m : T \in p$, $\exists \hat{}m : T' \in q$ such that $T <: T'$.

   (c) $q <: p$ if:

   - $\mathcal{T}(q) \subseteq \mathcal{T}(p)$.
   - $\forall\, m : T \in q$, $\exists m : T' \in p$ such that $T' <: T$.
   - $\forall\, \hat{}m : T \in q$, $\exists \hat{}m : T' \in p$ such that $T <: T'$.

   (d) If $p <: q$ and $q <: p$ then $p = q$.

2. Otherwise, follow these steps:

   (a) Find $\mathcal{T}(\overline{p})$ and $\mathcal{T}(\overline{q})$.

   (b) Find $N_{max}(\overline{p})$ and $N_{max}(\overline{q})$.

   (c) Create the sets $\Psi = \bigcup_{n=1}^{N_{max}(\overline{q})} \mathcal{T}(\overline{p})^n$ and $\Psi' = \bigcup_{n=1}^{N_{max}(\overline{p})} \mathcal{T}(\overline{q})^n$.

   (d) $q <: p$ if $\mathcal{T}(\overline{q}) \subseteq \Psi$ and,

- $\forall\ m : T \in q$, $\exists m : T' \in p$ such that $T' <: T$.
- $\forall\ \hat{}m : T \in q$, $\exists \hat{}m : T' \in p$ such that $T <: T'$.

(e) $p <: q$ if $\mathcal{T}(\overline{p}) \subseteq \Psi'$ and,

- $\forall\ m : T \in p$, $\exists m : T' \in q$ such that $T' <: T$.
- $\forall\ \hat{}m : T \in p$, $\exists \hat{}m : T' \in q$ such that $T <: T'$.

(f) (e) and (d) hold true then $p = q$. ■

Similar to proof for protocols equality, this algorithm is also both *sound* and *complete.*

Following rules are direct conclusions of the definition of protocols satisfaction. $\bigvee p_i$ stands for the alternation $p_1 \mid p_2 \mid \cdots$ and $\bigwedge p_i$ to stand for the sequence $p_1; p_2; \ldots$.

1. For repetition, we have the obvious rules:

$$\epsilon <: *p$$

$$\epsilon <: ?p$$

$$p <: *p.$$

$$p <: ?p.$$

$$p <: +p.$$

2. For alternation, a protocol is satisfied by any protocol with a non-empty subset of alternatives. Formally:

$$\bigvee_{i \in R} p_i <: \bigvee_{j \in S} p_i \quad \text{if } R \neq \emptyset \text{ and } R \subseteq S.$$

3. The subprotocol relation extends over each operator:

$$\frac{\Gamma \vdash p <: p'}{\Gamma \vdash *p <: *p'} \quad [1]$$

$$\frac{\Gamma \vdash p_i <: p'_i \text{ for } i \in S}{\Gamma \vdash \bigwedge_{i \in S} p_i \ <: \ \bigwedge_{i \in S} p'_i} \quad [2]$$

$$\frac{\Gamma \vdash p_i <: p'_i \text{ for } i \in S}{\Gamma \vdash \bigvee_{i \in S} p_i \ <: \ \bigvee_{i \in S} p'_i} \quad [3]$$

**Theorem 4** *There exists a minimum ("bottom") protocol, $\perp_p$, such that, for any given protocol p, $\perp_p <: p$.*

*Proof:* Define $\perp_p$ equal to $\perp_p = \phi$. Therefore, for any given protocol $p$:

$$\begin{aligned} \mathcal{T}(\perp_p) &= \mathcal{T}(\phi) = \{\ \} \\ &\Rightarrow \mathcal{T}(\perp_p) \subseteq \mathcal{T}(p) \\ &\Rightarrow \perp_p <: p \end{aligned}$$

■

**Theorem 5** *There exists a maximum ("Top") protocol, $\top_p$, such that, for any given protocol p, $p <: \top_p$.*

*Proof:* For any variable identifier $x_i$ defined in our enviroment, define $\top_p$ as equal to $\top_p = *[(\bigvee x_i : \ \texttt{any}\ ) \mid (\bigvee \hat{}\, x_i : \ \texttt{any}\ )]$ where $\bigvee xi$ means the union of all variable names. Therefore, for any given protocol $p$, it is obvious that $p <: \top_p$. Same as the Top type "**any**" this protocol is just useful in theory. ■

## 5.3 Ports

The features we have defined up to now - function types, record types, pair and tuple types, reference types, protocol types, and subtyping- are sufficient to build up a collection of programming idioms supporting objects, ports and closures. Therefore, in the following sections we will introduce these concepts and we will implement their behaviors by using the types and the type rules we've defined recently.

A cell accesses protocols within *ports*. A port declaration introduces a port name and associates a protocol with it. The syntaxe of ports are given in Table 38. Note that the symbol '+:' indicates that the port is associated with a process or a cell that *provides* to protocol, that is, it is a *server*. The symbol '-:' indicates that the port is associated with a process or a cell that *needs* to protocol, that is, it is a *client*. Moreover, the symbol '::' indicates that no particular direction is associated with the port; it is used to declare a port that *links* a server to a client.

| *PortDeclaration* | = | PortName ('+:' \| '-:' \| '::') *Protocol* |
|---|---|---|
| t::= | | terms |
| | t+:t | port provider |
| | t-:t | port need |
| | t::t | port link |
| | t.$\alpha$ | projection |

Table 38: Syntax Of Ports In Erasmus

The type of a port is a Cartesian product, pair type, of $A \times B$ where $A$ belongs to the set PORT-TYPE and $B$ is a protocol type. The set PORT-TYPE consists of two types. These two types are PORT-PROVIDER and PORT-NEED and are the keywords of the grammar. The only defined value for these is *zero*, and the following subtype relation is defined on these two types.

$$\texttt{PORT-NEED} <: \texttt{PORT-PROVIDER}$$

The symbols $+:$ and $-:$ in the syntax of ports are functions that generate ports. These two functions can be described by the first order lambda calculus as follows.

$$+: \;=\; \lambda p : [\alpha_1 : T_1, \ldots \alpha_n : T_n] .(< 0 : \texttt{PORT-PROVIDER}, p >)$$

$$-: \;=\; \lambda p : [\alpha_1 : T_1, \ldots \alpha_n : T_n] .(< 0 : \texttt{PORT-NEED}, p >)$$

Therefore, $v+ : p$ and $v- : p$ are abbreviations for $v : (+:)(p)$ and $v : (-:)(p)$, and they have the following types.

$$(+:) \;:\; [\alpha_1 : T_1, \ldots \alpha_n : T_n] \rightarrow < \texttt{PORT-PROVIDER}, [\alpha_1 : T_1, \ldots \alpha_n : T_n] >$$

$$(-:) \;:\; [\alpha_1 : T_1, \ldots \alpha_n : T_n] \rightarrow < \texttt{PORT-NEED}, [\alpha_1 : T_1, \ldots \alpha_n : T_n] >$$

If $v$ is a port then the indexing operator could be explained as follows:

$$v.\alpha_i \;=\; \pi_2(v).\alpha_i$$

As mentioned above, ports have pair types, therefore the subtyping rule for ports are the same as the subtyping rule for pair types. Therefore, for any given protocol $p_1$ and $p_2$, the following rule is trivial.

$$\frac{\Gamma \vdash p_1 <: p_2}{\Gamma \vdash (v- : p_1) <: (v- : p_2)} \quad \text{[T-SUB-PORTS-1]}$$

$$\frac{\Gamma \vdash p_1 <: p_2}{\Gamma \vdash (v+ : p_1) <: (v+ : p_2)} \quad \text{[T-SUB-PORTS-2]}$$

$$\frac{\Gamma \vdash p_1 <: p_2}{\Gamma \vdash (v- : p_1) <: (v+ : p_2)} \quad \text{[T-SUB-PORTS-3]}$$

Moreover, if $v$ is a port then the indexing operator could be explained as follows:

$$v.\alpha_i \quad = \quad \pi_2(v).\alpha_i$$

The link operator :: is a function that links closures together. Table 39 below shows the appropriate type rules for the ports.

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]}{\Gamma \vdash v+ : p} \quad \text{[T-PORT PROVIDER]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]}{\Gamma \vdash v- : p} \quad \text{[T-PORT CLIENT]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n]}{\Gamma \vdash v :: p} \quad \text{[T-PORT LINK]}$$

$$\frac{\Gamma \vdash p : [\alpha_1 : T_1, \ldots, \alpha_n : T_n], \Gamma \vdash v+ : p}{\Gamma \vdash v.\alpha_i : \texttt{ref } T_i} \quad \text{[T-VAL PORT]}$$

$$\frac{\Gamma \vdash p_1 <: p_2}{\Gamma \vdash (v_1- : p_1) <: (v_2+ : p_2)} \quad \text{[T-SUB PORT]}$$

Table 39: Type Rules for Port in Erasmus

Note that the type rule "T-Port Provider" indicates that if variable $p$ has a protocol type then v:+p is a well formed term in our static environment. This holds true for the type

rules "T-Port Client" and "T-Port Link". Moreover, the type rule "T-Val Port" indicates that if variable $p$ is a protocol type with a field $\alpha : T$ then v.$\alpha$ is a well formed term and has a type `ref` $T$ in our static environment.

## 5.4 Closures

As mentioned before, an execution of the Erasmus project comprises cells and their interactions. Each cell may have some processes that describe the behavior of a cell. A *closure* is a process that may be parameterized by variables and ports. The port declarations come first, followed by a bar ("|"), and then a *sequence* that determines the behavior of the process.

$$Closure = \mid \text{`\{'} \{Declaration\} \text{;`|'} Sequence \text{`|'}.$$

when the closure is instantiated, it is given arguments corresponding to its parameters.

$$Instantiation = (\ Cell \mid Closure\ )\ \text{`('}\ \{PortName|VarName\},\ \text{`)'}$$

A closure can be passed around, even sent across a network, and will perform as advertised provided that it is given appropriate arguments. The types defined in the previous sections are sufficient to describe the closures. As an example a closure without a port and with ports could be explained as:

$$
\begin{aligned}
\{|S\} &= \lambda\_ : \texttt{Void}.S : \texttt{Void} \rightarrow \texttt{Void} \\
\{v+ : p \mid S\} &= \lambda x : \texttt{<Port-Provider, Protocol\_Type>}.S \\
&\quad : \texttt{<PORT-PROVIDER, Protocol\_Type>} \rightarrow \texttt{Void} \\
\{v- : p \mid S\} &= \lambda x : \texttt{<PORT-NEED, Protocol\_Type>}.S \\
&\quad : \texttt{<PORT-NEED, Protocol\_Type>} \rightarrow \texttt{Void}
\end{aligned}
$$

Therefore, closures are fucntions that accept ports as their arguments and execute their body. Thus, a closure in the form of $c = \{A : \texttt{T} \mid S : \texttt{Void}\}$ is an abbreviation for $\lambda A : \texttt{T}.S$ which has a function type, and could be explained by *bounded universally quantified types.* Table 40 given below shows the appropriate type rules for closures.

$$\frac{\Gamma \vdash S : \texttt{Void}}{\Gamma \vdash \{|S\} : \texttt{Void} \rightarrow \texttt{Void}} \quad \text{[T-Closure Without Ports]}$$

$$\frac{\Gamma \vdash \texttt{T} <: \texttt{<PORT-PROVIDER, } \top_p\texttt{>}, \Gamma \vdash S : \texttt{Void}}{\Gamma \vdash \{v : T \mid S\} : (\forall T' <: \texttt{<PORT-PROVIDER,} \top_p\texttt{>}).\texttt{T'} \rightarrow \texttt{Void}} \quad \text{[T-Closure-With-Port]}$$

$$\frac{\Gamma \vdash C : (\forall T <: \texttt{<PORT-PROVIDER,} \top_p\texttt{>}).\texttt{T} \rightarrow \texttt{Void}}{\Gamma \vdash C(T)} \quad \text{[T-Closure-Inst]}$$

Table 40: Type Rules for Closures in Erasmus

As shown in Table 40 above, a closure has a function type. Therefore, the subtyping rule for closures is same as the subtyping rule for functions. Note that the behavior of a closure that accepts both ports and variables as its arguments could be explained same as above by using the *bounded universally quantified types.*

## 5.5 Cells

To summarize so far, we have constructed our type system by introducing some appropriate types such as basic types, record types, function types, protocol types and etc. By using these types we explained the types of ports and closures. In this section we will introduce a new concept called cells, and we will concentrate on explaining the behaviors of cells by using the features we have introduced in the previous chapters. Moreover, some appropriate examples and diagrams are also given to make it easier to understand these behaviors.

As mentioned earlier, an execution of Erasmus language comprises cells and their interactions. A cell could be as small as a single character or it could be as large as a distributed system. The behaviors of cells are explained by their processes. Cells and their processes may communicate with each other through their protocols. Protocols are expressions which specify the structure of messages as well as their allowable sequences. We also mentioned that cells access protocols within ports.

The syntax of a cell in the Erasmus language is given below:

$$Cell = CellName|\text{`('}\{Declaration\}\text{`|'}\{Declaration|Instantiation\}\text{`)'}$$

$$Instantiation = (\ Cell\ --\ Closure)\ \text{`('}\{PortName|VarName\}\text{`)'}$$

Note that the port and variable declarations written before the bar are from outside the cell.

Cells in the Erasmus project could be regarded as object generators. A cell is just a function that generates data structure encapsulating some internal *state* and offering access to this state via a collection of methods. The internal state is typically organized as a

number of mutable *instance variables* (or fields) that are shared among the methods and inaccessible from the outside of a cell.

Following are the examples in which we will explain the behaviors of cells by using the features discussed before.

### 5.5.1 Example 1: "Hello World!"

The simplest possible Erasmus program consists of a process definition

```
proc = { | stdout := "Hello, world!" };
```

a cell definition that instantiates the process

```
cell = ( proc() );
```

and an instantiation of the cell

```
cell();
```

In this example `cell` is a function that first generates an object with the method `proc` and then instantiates this method which prints the "Hello World!" on the screen. By using the features discussed before we are able to describe the behavior of this cell. Therefore, the type of `cell` could be explained as:

$$
\begin{array}{lcl}
F_1 & : & \texttt{TYPE(proc)} \\
 & : & \texttt{Void} \rightarrow \texttt{Void} \\
F_2 & = & \lambda x : F_1.\{y = x\} \\
 & : & (\texttt{Void} \rightarrow \texttt{Void}) \rightarrow \{y : F_1\} \\
F_3 & = & \lambda x : F_1.(F_2(x).y()) \\
 & : & F_1 \rightarrow \texttt{Void} \\
\texttt{cell} & = & \lambda\_ : \texttt{Void}.(F_3\ \texttt{(proc)}\ ) \\
 & : & \texttt{Void} \rightarrow F_3
\end{array}
$$

$F_1$ is the type of the closure `proc`, and $F_2$ is a function that accepts a method and generates a record containing that method. Moreover, $F_3$ is a function that first accepts a method then generates a record containing that method and then instantiates that method. Note that how we use the first order lambda calculus to bind `proc` in the record. Therefore, `cell` could be defined as a function that instantiates $F_3$ with `proc`.

### 5.5.2 Example 2: Standard Input and Output

In this example we try to describe the behavior of a cell that links its two closures that wish to communicate. The keyboard and screen servers combine to give a full interface for simple tests. Here are the protocols.

```
kbprot = [ *(^kbd: Text) ];
scprot = [ *(scr: Text) ];
```

`kbprot` is a protocol that sends the reply message `^kbd` of type `Text` zero or more times, and `scprot` is a protocol that sends the message `scr` of type `Text` zero or more times.

And the keyboard server is:

```
keyboard = { p +: kbprot |
  loop
    p.kbd := stdin
  end
};
```

`keyboard` is a process which has a port `p` as its argument. This process can access the protocol `kbprot` through the port `p`. The port `p` is a port provider which indicates that it provides something. The body of this process is a loop which reads the input screen and stores it in `p.kbd`.

The screen server is similar:

```
screen = { p +: scprot |
  loop
    stdout := p.scr
  end
};
```

Similarly `screen` is a process which has a port `p` as its argument. This process can access the protocol `scprot` through the port `p`. The port `p` is a port provider which indicates that it provides something. The body of this process is a loop which prints the values stored in `p.scr`.

The following process uses the keyboard and screen servers to conduct a dialog with the user:

```
dialog = { pk -: kbprot; ps -: scprot |
  loop
    ps.scr := "What is your name? ";
    name: Text := pk.kbd;
    ps.scr := "Hello, " + name + "!\n";
  end
};
```

dialog is a process which has a port pk and a port ps. This process can access the protocol scprot and kbprot through the ports ps and pk. These ports are port need which indicate that they need something. The body of this process is a loop that first sends a Text message, "What is your name", to the screen by using the ps port, and then waits to read a Text message from pk port to store it in variable name, and at the end sends a message, "Hello, + name + ! " by using the ps port.

The **main** cell runs all of these processes:

```
mainCell = (
  pk :: kbprot; ps :: scprot;
  keyboard(pk); screen(ps); dialog(pk, ps);
);

mainCell();
```

mainCell is a cell which has two link ports pk and ps, and three processes keyboard, screen, and dialog. The function mainCell() instantiates the cell which links processes

that share common port(s). In this example, `mainCell()` link `keyboard` and `screen` processes to the `dialog` process. Note that only those processes which share common port(s) could be linked together.

Programs with several protocols and processes are easier to understand if accompanied with a diagram. Figure 7 is a diagram corresponding to the program discussed in this section. The enclosing cell is drawn as a thick blue outline.[1] The outline has no ports, corresponding to the fact that a complete Erasmus program is a "closed world".

The processes are drawn as red rectangles. Their boundaries contain ports labelled "+" for servers and "−" for clients. Conventionally, we put the local port names inside the process boxes and label the connecting link with the protocol. The names that the cell uses to link the processes are not shown, although they could be.

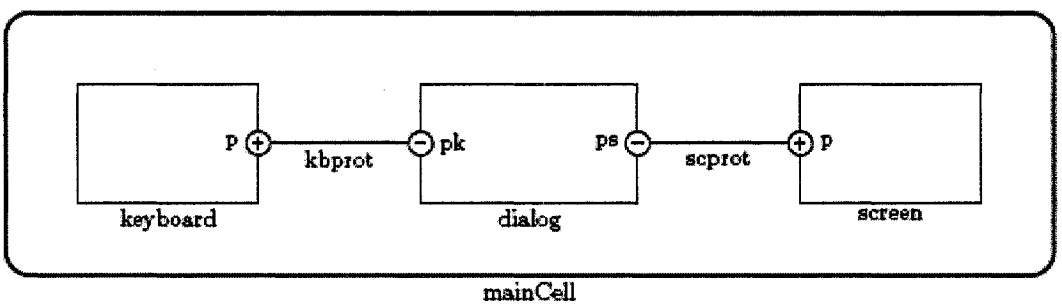


Figure 7: Communicating with the Keyboard and Screen

Following is the explanation for the behavior of this program. As usual consider $S$ as a term which has type `Void`.

[1] If this document is printed without colour, you can recognize cell outlines because they are thicker and have rounded corners.

The types of the closures are:

$$\texttt{keyboard} = \lambda x : \texttt{<PORT-PROVIDER,}[*(\hat{}kbd : Text)]\texttt{>.S}$$

$$: \quad \texttt{<PORT-PROVIDER,}[*(\hat{}kbd : Text)]\texttt{>} \rightarrow \texttt{Void}$$

$$\texttt{screen} = \lambda x : \texttt{<PORT-PROVIDER,}[*(scr : Text)]\texttt{>.S}$$

$$: \quad \texttt{<PORT-PROVIDER,}[*(scr : Text)]\texttt{>} \rightarrow \texttt{Void}$$

$$\texttt{dialog} = \lambda x : \texttt{<PORT-NEED,}[*(\hat{}kbd : Text)]\texttt{>.}(\lambda x : \texttt{<PORT-NEED,}[*(scr : Text)]\texttt{>.S})$$

$$: \quad \texttt{<PORT-NEED,}[*(\hat{}kbd : Text)]\texttt{>} \rightarrow (\texttt{<PORT-NEED,}[*(scr : Text)]\texttt{>} \rightarrow \texttt{Void})$$

and the type of the **main** cell that runs all of these processes is: (TP(t) stands for the type of the term t)

$$F_1 = \lambda x : \texttt{<TP(keyboard),TP(screen),TP(dialog)>.}$$

$$\{x_1 = \pi_1(x), x_2 = \pi_2(x), x_3 = \pi_3(x)\}$$

$$: \quad < t_1, \ldots, t_n > \rightarrow \{\alpha_1 : \texttt{TP(keyboard)}, \alpha_2 : \texttt{TP(screen)}, \alpha_3 : \texttt{TP(dialog)}\}$$

$\texttt{mainCell} = \lambda\_\_ : \texttt{Void.}$

(

Link(keybord,dialog);

Link(screen,dialog);

$F_1$(<keyboard,screen,dialog>).$x_1$ (pk)

$F_1$(<keyboard,screen,dialog>).$x_2$ (ps)

$F_1$(<keyboard,screen,dialog>).$x_3$ (pk,ps)

)

$: \quad \texttt{Void} \rightarrow \texttt{Void}$

Therefore, `mainCell` links and instantiate the closures.

### 5.5.3 Link

As shown in the Figure 7, two closures and cells could link together and communicate. The syntax "::" in a port definition is used to indicate a link port which links closures and cells together. As mentioned in the section portI, we defined the type of a port as a Cartesian product, pair type, of $A \times B$ where $A$ belongs to the set `PORT-TYPE` and $B$ is a protocol type. We also mentioned that the set `PORT-TYPE` consists of two types `PORT-PROVIDER` and `PORT-NEED`. It is useful in this place to add a new type called `PORT-LINK` to this set with the following attributes.

- it is the keyword of the grammar.
- the only defiend value for this type is *zero*.
- `PORT-LINK <:PORT-NEED <:PORT-PROVIDER.`

Therefore, the type of a link port $v :: p$ is $v :< \texttt{PORT-LINK, TYPE(p)} >$.

Following are the conditions in which two closures could be linked. In the following $C_1$ and $C_2$ are closures.

1. $C_1$ and $C_2$ must share a common link port, and must accept the link port as their argument.
2. $C_1$ must be a client closure. (it must contain a port need)
3. $C_2$ must be a server closure. (it must contain a port provider)
4. the type of $C_1$ must be a subtype of the type of $C_2$ that is, $C_1 <: C_2$.

Following is an example in which `mainCell()` links two closures `A` and `B` together and runs the program. As usual the types of $S$ and $S'$ are `Void`, and $p$, $p'$, and $p''$ are protocols.

```
A = { v +: p | S}


B = { v' -: p' | S'}


mainCell = ( p1 :: p'' ; A(p1) ; B(p1););


mainCell();
```

According to the conditions above $A$ and $B$ could be linked together if the following steps hold true.

1. `p''` <:`p` which implies `p1` <: `v`.
2. `p''` <:`p'` which implies `p1` <: `v'`.
3. `v'` <:`v` which implies `B` <: `A`.

# Chapter 6

# Conclusion and Future Work

The Erasmus language discussed in this document is being developed by Peter Grogono at Concordia University and Brian Shearing at The Software Factory in England. The Erasmus language is a process oriented language which is mainly based on cells and their interactions.

We constructed a suitable type system for this programming language which makes it to be both safe and well-typed. We started by describing some fundamental concepts such as the notion of type in programming languages, the need for the existence of a type system, and the different ways in which a type system can be constructed. The evolution of type languages from untyped universes to typed universes was also reviewed.

Some notations commonly used for describing type systems such as *judgments*, which are formal assertions about the typing of programs, *type rules*, which are implications between judgments, and *derivations*, which are deductions based on type rules were also explained.

The $\lambda$-calculus is used for the type system of this programming language. We started with the first order lambda calculus and augmented our type system with a broad spectrum

of simple types, record types, function types, map and array types, reference types. We also formalized our type system by providing appropriate type rules with subtyping rlues for these types.

Universal quantification was introduced to model parametric polymorphism. Accordingly, we augmented our type system with the universally quantified types which enrich our type system to model generic functions with type parameters. We also augmented our type system with bounded universal quantification which enriches our type system by providing explicit subtype parameters. We used the second order lambda calculus to describe the behavior of these concepts.

Messages, protocols, ports, closures, and cells which are the main heart of the Erasmus language were also explained, and we augmented their type rules with their subtyping relationships into our previous type system. Protocols satisfaction was introduced to indicate a relationship between compatible protocols. We introduced traces sets and used them to compare protocols (protocols satisfaction). An algorithm was also given for this reason, and we proved that this algorithm is both sound and complete. The second order lambda calculus was used to explain the behaviors of these concepts and their sutyping relationships in a consistent framework.

A link function was also introduced to explain the behaviors of cells during the communication periods. We also explained the conditions in which two (or more) cells are able to link together and communicate, but it needs future works. The syntax of the Erasmus language was also explained while we were constructing the type system. We also provided some examples to clarify how these concepts could join together and make an Erasmus program.

**Future Work** In conventional typed languages, the compiler assigns a type to every expression and sub expression. However, the programmer does not have to specify the type of every sub expression of every expression: type information need only be placed at critical points in a program, and the rest is deduced from the context. This deduction process is called *type inference.* Typically, type information is given for local variables and for function arguments and results. The type of expressions and statements can then be inferred, given that the type of variables and basic constants is known.

Type inference is usually done *bottom-up* on expression trees. Given the type of the leaves (variables and constants) and type rules for the ways of combining expressions into bigger expressions, it is possible to deduce the type of whole expressions. For this to work it is sufficient to declare the type of newly introduced variables. Note that it may not be necessary to declare the return type of a function or the type of initialized variables.

The ML language introduced a more sophisticated way of doing type inference. In ML it is not even necessary to specify the type of newly introduced variables. The type inference algorithm still works bottom-up. The type of a variable is initially taken to be unknown. The instantiation of type variables is done by Robinson's unification algorithm [10], which also takes care of propagating information across all the instances of the same variable, so that incompatible uses of the same variable are detected. Introductory exposition of polymorphic type inference can be found in [4].

The best type inference algorithm known is the one used in ML and similar languages. This amounts to saying that the best we know how to do is type inference for type systems with little existential quantification, no subtyping, and with a limited (but quite powerful) form of universal quantification. Moreover, in many extensions of the ML type system the

type-checking problem has been shown to be undecidable.

Type inference reduces to type checking when there is so much type information in a program that the type inference task becomes trivial. More precisely we can talk of type checking when all the type expressions involved in checking a program are already explicitly contained in the program text, i.e., when there is no need to generate new type expressions during compilation and all one has to do is match existing type expressions.

We probably cannot hope to find fully automatic type-inference algorithms for the type system we have presented in this document. There is actually one problem, which is however shared by all polymorphic languages, and this has to do with type checking side-effects. Some restrictions have to be imposed to prevent violating the type system by storing and fetching polymorphic objects in memory locations. Examples can be found in [12] and [2]. There are several known practical solutions to this problem [8, 15] which trade off flexibility with complexity of the type checker.

# Bibliography


[1] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] A. Albano, L. Cardelli, and R. Orsini. A Strongly Typed, Interactive Conceptual Language, Transactions on Database Systems. *Journal of the ACM*, 12(1):230–260, June 1985.

[3] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. *F-bounded polymorphism for object-oriented programming.* ACM Press, Addison-Wesley, 1989.

[4] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[5] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

[6] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies).* Princeton University Press, Princeton, NJ, USA, 1985.

[8] Luis Manuel Martins Damas:. Type Assignment in Programming Languages. PhD thesis, University of Edinburgh. 1985.

[9] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[10] J.A.Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–49, Jan 1996.

[11] P. J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965.

[12] M. Gordon and R. Milner and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer, 1979.

[13] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. Technical Report 86, 1989.

[14] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[15] R.Milner:. A proposal for Standard ML, Proc. Symposium on Lisp and Functional Programming, Austin, Texas, . *Journal of the ACM, New York*, pages 184–197, August 6-8 1984.

[16] Anthony J. H. Simons:. The Theory of Classification, Part 1: Perspectives on Type Compatibility. *Journal of Object Technology*, 1(1):55–61, May-June 2002. www.jot.fm/-issues/issue_2002_05/column5.

[17] Anthony J. H. Simons:. The Theory of Classification, Part 4: Object Types and Subtyping. *Journal of Object Technology*, 1(5):27–35, November-December 2002. www.jot.fm/-issues/issue_2002_11/column2.

[18] Anthony J. H. Simons:. The Theory of Classification, Part 5: Axioms, Assertions and Subtyping. *Journal of Object Technology*, 2(1):13–21, January-February 2003. www.jot.fm/issues/issue_2003_01/column2.

[19] Anthony J. H. Simons:. The Theory of Classification, Part 6: The Subtyping Inquisition. *Journal of Object Technology*, 2(2):17–26, March-April 2003. www.jot.fm/issues/-issue_2003_03/column2.

[20] Anthony J. H. Simons:. The Theory of Classification, Part 7: A Class is a Type Family. *Journal of Object Technology*, 2(3):13–22, May-June 2003. www.jot.fm/issues/-issue_2003_05/column2.

[21] Anthony J. H. Simons:. The Theory of Classification, Part 8: Classification and Inheritance. *Journal of Object Technology*, 2(4):55–64, July-August 2003. www.jot.fm/-issues/issue_2003_07/column4.

[22] C. Strachey. *Fundamental Concepts In Programming Languages.* August 1967. Lecture Notes for International Summer School in Computer Programming, Copenhagen.