UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

A CONCURRENT PROGRAMMING LANGUAGE WITH SESSION TYPES

Juliana Patrícia Vicente Franco

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

A CONCURRENT PROGRAMMING LANGUAGE WITH SESSION TYPES

Juliana Patrícia Vicente Franco

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos
e co-orientado pelo Prof. Doutor Francisco Cipriano da Cunha Martins

2013

Agradecimentos

Agradeço ao meu orientador, professor Vasco Vasconcelos, pela constante disponibilidade, apoio e pela oportunidade que me deu para fazer um projeto numa área tão interessante. Foi incansável durante o desenvolvimento deste trabalho de modo a torná-lo o melhor possível. Foi sem dúvida um ano em que aprendi bastante e grande parte do que aprendi foi graças a ele. Muito obrigada professor. Tem sido uma excelente experiência trabalhar consigo!

Não posso deixar de agradecer à minha família pois sem ela não teria sido possível alcançar esta meta. Aos meus pais, Júlio e Ana, agradeço todo o apoio e incentivo que sempre me deram e também pela educação que me proporcionaram e que me permitiu chegar até aqui. Agradeço também ao meu irmão, Flávio, por me ter feito ver que Engenharia Informática era o curso certo para mim e à minha irmã, Cassandra, porque mesmo não tendo idade para perceber o porquê de tanto trabalho, nunca me deixou desanimar. A vocês os quatro, muito obrigada. Foram essenciais!

Por fim, mas não menos importante, quero agradecer a todas as pessoas, que ao longo do meu percurso na FCUL me ajudaram a alcançar os meus objetivos. Em especial a três pessoas que me acompanharam e tornaram tudo mais fácil desde os primeiros dias de faculdade. À Rita Henriques pela companhia em muitos fins de semana e noites de trabalho, e por uma amizade que tornou tudo mais fácil. E também à Hélia Grilo e à Mafalda Gomes por toda a amizade e apoio que me deram ao longo destes cinco anos e, principalmente, nesta recta final. Foram muito importantes no caminho até aqui!

Aos meus pais e irmãos.

Resumo

Em computações concorrentes complexas onde os processos comunicam por troca de mensagens existe normalmente um elevado número de mensagens trocadas. Geralmente, estes programas são muito difíceis de implementar visto que a ordem e o tipo das mensagens enviadas e recebidas são muitas vezes alvos de erros dos programadores.

Numa comunicação entre dois processos que partilham o mesmo meio de comunicação é necessário garantir que estes trocam as mensagens de forma correcta: quando um envia um valor de tipo inteiro, o outro deve estar a espera de receber um valor de tipo inteiro, quando um oferece um conjunto de opções, o outro deve estar pronto a selecionar uma dessas opções e quando um tem de terminar a sua interação nesse canal, o outro também deve terminar. Também é importante garantir a não ocorrência de condições de corrida, assegurando que durante uma comunicação entre dois intervenientes, outros não interferem. Por isso, é útil abstrair protocolos que governam as interações entre os intervenientes de uma comunicação descrevendo quando e quais são as mensagens trocadas entre eles. Temos então os tipos de sessão que descrevem as contínuas interações entre os diferentes parceiros de uma comunicação.

Para definir quantos parceiros conhecem um dado canal de comunicação em qualquer ponto do programa, é possível qualificar o tipo de sessão, associado a esse canal, como linear ou partilhado. Os tipos qualificados como lineares representam extremidades de canais que apenas podem aparecer em exatamente um fio de execução, enquanto que os partilhados representam extremidades que podem ocorrer num número ilimitado de fios de execução. Deste modo podemos garantir que não existem condições de corrida, pois em situações sujeitas a tal pode-se usar tipos lineares. Mas por outro lado, se for necessário que muitos processos tenham acesso ao canal podemos usar tipos partilhados.

Com vista a facilitar a programação concorrente, construímos uma nova linguagem de programação chamada SePi. Esta é uma linguagem de programação concorrente baseada no cálculo pi que usa tipos de sessão linearmente refinados para descrever as operações realizadas nos canais de comunicação. Esta linguagem concorrente, onde os processos comunicam de forma síncrona através canais bidirecionais (cada canal é definido por duas extremidades), permite que as interações entre processos sejam verificadas em tempo de compilação utilizando os tipos de sessão para descrever o tipo e a ordem das mensagens, bem como o número de processos que podem partilhar os canais.

A linguagem SePi tem construtores de processos para enviar e receber uma mensagem, para representar a receção replicada, a composição paralela, com zero ou mais processos a serem executados concorrentemente, o processo condicional e a criação de novos canais definidos por duas extremidades. Os processos podem enviar mensagens de tipo inteiro, booleano, *string* ou extremidades de canais. Para além dos tipos primitivos, a linguagem apresenta também tipos para descrever o envio e receção de mensagens, a oferta e a seleção de opções, um tipo para representar canais onde não é possível ocorrer mais interações, tipos refinados e tipos recursivos. Por exemplo o tipo **lin**?**boolean**.**lin**!**integer**.**end** representa uma extremidade de um canal que recebe um valor booleano e de seguida envia um inteiro. Os tipos recursivos permitem efetuar a mesma operação, ou o mesmo conjunto de operações, no mesmo canal um número indeterminado de vezes. Os tipos refinados, uma forma de tipos dependentes, aparecem com o objetivo de especificar certas propriedades dos valores de um programa SePi anexando fórmulas a tipos. Estas fórmulas podem ser predicados não interpretados, na forma de p(v1, ..., vn), tensores na forma de A x A ou **unit**. Por exemplo, um cliente que queira efetuar um pagamento a uma loja utilizando o seu cartão de crédito quer ter a certeza que este é utilizado apenas uma vez e para cobrar a quantia certa. Os tipos de sessão não são suficientes para assegurar este comportamento, pois podemos garantir que a loja cobra um inteiro, mas não podemos garantir que cobra o montante exato da compra. No entanto podemos usar tipos refinados para especificar o montante que vai ser cobrado no cartão de crédito e o número de vezes que esta cobrança é feita. Um exemplo de tipos refinados é então {x:**integer** | charge(x, card)}. Para conseguirmos obter este comportamento, tratamos das fórmulas como se fossem recursos. Elas são introduzidas no sistema de tipos através do processo *assume*, passadas entre processos através dos tipos refinados e removidas através do processo *assert*.

Existe outro conceito muito importante na nossa linguagem, a dualidade, que nos permite garantir que dois processos que partilham um canal comportam-se de forma complementar. Dizemos então que as duas extremidades de um canal têm tipos duais ou complementares. Isto é, quando uma extremidade é usada para enviar uma mensagem a outra deve ser usada para receber, quando uma é usada para selecionar uma opção então a outra tem de ser usada para oferecer um conjunto de opções. O tipo que define canais sem interação é dual dele próprio e os tipos primitivos e refinados não têm a função de dualidade definida. Por exemplo o tipo **lin**?**boolean**.**lin**!**integer**.**end** é dual de **lin**!**boolean**.**lin**?**integer**.**end**.

Até agora descrevemos apenas a linguagem "base". A versão mais recente da linguagem SePi é baseada na primeira mas apresenta mais construtores, tais como construtores derivados e abreviaturas. Como abreviaturas temos o uso opcional do qualificador linear ou o uso de um * para representar uma classe comum de tipos partilhados, o operador **dualof** que permite obter o tipo dual de outro, e a composição paralela de zero valores que

também é opcional. Como construtores derivados dos da linguagem base, temos o envio e receção de múltiplos valores, a definição de processos (que deriva de uma criação de um novo canal seguido de uma receção replicada em paralelo com o resto do programa) e a declaração de tipos mutuamente recursivas. A linguagem SePi também apresenta expressões binárias, como as aritméticas, lógicas e relacionais, ou expressões unárias, como a negação.

O nosso objetivo ao introduzir estes novos construtores é o de capturar com uma sintaxe especial, os idiomas e padrões de programação mais comuns e assim reduzir o número de linhas de código SePi e eventuais erros associados. Esta linguagem é baseada nos trabalhos de Vasconcelos [36] e Baltazar et al. [3]. Para facilitar a programação em SePi desenvolvemos um plugin para o Eclipse de modo a permitir a validação sintática e semântica, completação e *refactoring* de código, bem como interpretação. Para além disso, implementamos também uma versão para a linha de comandos.

A nossa implementação foi feita utilizando a *framework* Xtext que permite o desenvolvimento de novas linguagens de programação e plugins para o Eclipse. Sendo que o Xtext gera vários dos componentes de um compilador/interpretador, podemos dividir a nossa implementação em quatro partes: escrita da gramática, implementação de um mecanismo para verificar se todas as variáveis de um programa estão devidamente declaradas, implementação do algoritmo de verificação de tipos e escrita do interpretador. O interpretador é baseado na máquina abstrata de estados de Turner [32].

Este trabalho resulta assim numa linguagem de programação concorrente, baseada no cálculo pi monádico onde a comunicação entre os processos de um programa é governada por tipos que resultam de uma combinação entre tipos de sessão e tipos linearmente refinados.

Palavras-chave: Concorrência, tipos de sessão, tipos refinados, cálculo pi, canais de comunicação

Abstract

We present, SePi, a concurrent programming language based on the monadic pi-calculus, where communication among processes is governed by linearly refined session types. In SePi processes communicate synchronously, by message-passing, via bi-directional channels. Each communication channel is defined by two syntactically distinct end-points. Interactions on channels are statically verified against session types describing the type and order of messages exchanged, as well as the number of processes that may share a channel.

We start by designing a core language that includes constructs to send and receive messages (including a form of replicated input) and to select and offer a set of options, as well as parallel composition, conditional and channel creation. In order to allow describing more precise properties of programs, SePi further includes assume and assert primitives at the process level and refinements at the type level. Refinements are treated linearly, which allows a finer, resource-oriented use: each assumption is asserted exactly once, and conversely each assertion is also assumed exactly once.

On top of this core language we provide a few abbreviations and derived constructs with the purpose of facilitating code development, resulting in the current version of SePi. Abbreviations include the dualof operator and the optional type qualifiers; derived constructs comprise the input and output of multiple values and mutually recursive process definitions and type declarations.

SePi is currently implemented as an Eclipse plugin, allowing code development and interpretation with the usual advantages of an IDE, such as syntax highlighting, syntactic and semantic validation, code completion and refactoring.

Keywords: Concurrency, session types, refinement types, pi-calculus, communication channels

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 2

1.3 Deviations from the original plan thesis 3

1.4 Structure of the document 3

2 The pi-calculus, session types and related programming languages **5**

2.1 The pi-calculus 5

2.2 Session types 6

2.3 Session types in functional languages 7

2.4 Session types in object-oriented languages 8

2.5 Session types in imperative languages 10

2.6 Programming languages based on the pi-calculus 10

3 The SePi programming language **13**

3.1 The core language 13

3.2 The SePi language 16

3.2.1 Rationale 16

3.2.2 Introducing the language via an example 17

3.3 Algorithmic type checking 22

3.3.1 Well-formed types and formulae 23

3.3.2 Formulae normalisation, $\mathrm{normalise}\ A \mapsto \Gamma$ 23

3.3.3 Context normalisation, $\mathrm{normalise}\ \Gamma \mapsto \Gamma$ 23

3.3.4 Formula subtraction, $\Gamma \vdash A \mapsto \Gamma$ 24

3.3.5 Type equivalence, $\Gamma \vdash v\colon T \equiv T \mapsto \Gamma$ 24

3.3.6 Typing rules for expressions, $\Gamma \vdash e \mapsto T; \Gamma$ 24

3.3.7 Typing rules for processes, $\Gamma \vdash P \mapsto \Gamma; L$ 25

3.4 Derived constructs 29

3.5 Programming in SePi 32

3.5.1 A print server that makes sure values are printed in order 32

3.5.2 Channel forwarding 33

3.5.3 Request on a channel; respond on a distinct channel 35

4 Implementation **37**
4.1 Xtext and plugin implementation 37
4.2 The validation phase 39
4.2.1 The symbol table 40
4.2.2 Value hierarchy 41
4.2.3 Formulae hierarchy 41
4.2.4 Type hierarchy 42
4.2.5 The validation process 44
4.3 The interpreter 46
4.3.1 Machine states 46
4.3.2 The interpretation process 46
4.4 Metrics 49
4.5 Testing the compiler and the interpreter 49
4.6 Installing & running SePi 50

5 Conclusion **53**

Bibliography **58**

Chapter 1

Introduction

1.1 Motivation

In complex concurrent computations where processes communicate via message-passing, there is, a large number of exchanged messages. Message-passing programs are hard to implement correctly. The order and type of these messages are a common target of programmer mistakes. When two processes, P and Q are communicating, if P sends a message then Q must be ready to receive this message; if P sends an integer and after that, it is expecting to receive a string, then Q must be ready to receive an integer and then to send a string. A programmer can easily write a wrong program to describe the interactions between these two processes. For instance, if he writes that while one process sends an integer the other is prepared to receive a string (incorrect type of messages), or even, if he writes a program where one process sends a boolean value followed by a string value and the other receives first the string and then the boolean (wrong order of messages). Other kind of errors occur when multiple processes are trying to use the same resources. It may occur race conditions.

It is then useful to abstract protocols that govern the interactions between processes, describing when and what messages are sent or received by the communication parties. Session types appear in this context as a formalism to describe continuous interactions among different partners in a concurrent computation.

Furthermore, there are some properties about the messages exchanged by the processes that we want to ensure. For instance we may allow the programmer to define how many times a given value is used, or even constrain the name of the channels. We may look to the *Online store* example of Baltazar et al. [3]. In this example, there is an online store interacting with a bank and with multiple clients. Each client provides the store with the product that he wants to buy, his credit card and the product price. In turn, the store sends to the Bank the credit card and the amount in order to charge the client. The bank receives the credit card and the amount and proceeds with the charging. But what ensures that the store charges the exact amount, sent by the client, and that the credit card is used

to charge the agreed amount exactly once? Session types are not enough to ensure this behaviour (with session types we can describe that the store must send an integer to the bank, but we cannot tell that when a client sends an amount of `10`, the store must forward `10` to the bank, for example). The answer lies on refinement types—a form of dependent types which allow to specify properties of values in programs by attaching formulae to types.

There are multiple works that incorporates session types in functional [5, 27, 30] or object-oriented [8, 9, 23, 24] paradigms but, even being originally proposed for pi-calculus, there is no programming language pi-based that uses session types to describe the communication among the different partners of communication, where we may exercise examples, test program idioms and experiment with type systems. Besides, according with our knowledge, there is no language that combines the session types with linear refinement types.

What we propose is SePi, an exercise in the design and implementation of a concurrent programming language based on the pi calculus. The language features synchronous, bi-directional channel-based communication between concurrent processes. Processes use channel ends to read, write, offer a menu of choices or else select one such choice from a menu. There are also constructs to create a new channel and constructs to represent the parallel composition and the conditional process. At any point in a program, a channel end may be held by exactly one process or else shared by an unbounded number of processes. Interactions on channels are statically verified against session types describing the kind and order of messages exchanged, as well as the number of processes that may share a channel. In order to facilitate a more precise control on the properties of programs, SePi includes assume and assert primitives at the process level and refinements at the type level. Refinements are treated linearly, which allows a finer, resource-oriented use: each part of an assumption made with linear mode supports exactly one part of an assertion.

The formal foundation of the language can be found in references [3, 36]. On the top of this *core language* we introduced a number of abbreviations and derived constructs, in order to facilitate programming, such as, **dualof** operator, input/output of multiple values and mutually recursive process definitions and type declarations. These new constructs allows us to reduce the number of lines of code produced and the potentially associated errors. We implemented an interpreter based on the Turner's abstract machine [32]. In order to increase the productivity of the programmer, we created an Eclipse plugin to our language.

1.2 Contributions

The main contributions of this work can be summarized as follows:

- a new concurrent, message passing programming language based on the monadic

pi-calculus that features synchronous, bi-directional channel-based communication between concurrent processes.

- the implementation of a type system that combines session types and linear refinements to govern and describe interactions in concurrent programs.
- an Eclipse plugin that allows to develop with the usual advantages of an IDE, such as code completion, syntax highlighting, syntactic and semantic validation and interpreting SePi programs. We have also developed a SePi interpreter to be run from the command line.

From this work resulted the paper in reference [13].

1.3 Deviations from the original plan thesis

This section briefly compares what we planned to do in the beginning of this thesis and what we achieved. Initially we planned to implement the language as described in this thesis except for all aspects dealing with the formulae. Later on, we incorporate in SePi assume and assert processes, refinement types (and the formulae themselves). Other features of the language, such as derived constructs (input and output of multiple values, mutually recursive process definitions and type declarations), as well as some abbreviations, arisen naturally from the experience of programming in SePi. We estimate that these extensions implied an extra 3 months cost with respect to our initial schedule.

1.4 Structure of the document

The current chapter introduces our work, its motivations and contributions. The rest of the chapters are structured as follows:

Chapter 2 briefly reviews the notions of the pi-calculus and of session types, followed by programming languages that either incorporate session types or are based on the pi-calculus.

Chapter 3 presents the SePi language. First it describes the syntax of our core language and then it introduces the SePi language via the use of a running example of an *Online Donation Service*. The remaining sections describe the algorithmic type checking of the core language, the derived constructs that we introduced in the language and, finally, some examples written in SePi.

Chapter 4 describes how we have implemented our language and its Eclipse plugin, what classes we wrote to implement the validation and interpretation phases. It

also presents some metrics and how we have tested our work. The last section presents some informations about installing and running SePi.

Chapter 5 presents our conclusions and our plans for future work.

Chapter 2

The pi-calculus, session types and related programming languages

This chapter briefly reviews related work including the pi-calculus (Section 2.1) and concept of session types (Section 2.2), how session types can be incorporated in functional languages (Section 2.3), object-oriented languages (Section 2.4) and in imperative languages (Section 2.5). Finally we present two languages based on the pi-calculus (Section 2.6). The material in this chapter may be complemented with a recent survey [11].

2.1 The pi-calculus

Milner et al. [25, 26] introduce a calculus of concurrent communicating processes called the pi-calculus. Pi-calculus is a basic model of computation that uses a primitive notion of *interaction* based on reading and writing on channels. It can be used to model a network of interconnected processes that exchange messages and where messages may contain links to active processes. Figure 2.1 shows the syntax of processes of a variant in pi-calculus [36].

P ::=	Processes:
$x!vP$	send y along x
$x?y.P$	receive y along x
$x{*}?y.P$	replicated reception
$P_1 \mid P_2$	parallel composition
$(\nu ab)P$	scope restriction

Figure 2.1: Syntax of pi-calculus processes

Output process, of the form $x!vP$, writes on channel x the value v before continuing as process P. Input process, of the form of $x(y).P$, reads on channel x a value and binds it

to variable y before proceeding as P. Replicated reception provides for a persistent input. The syntax also introduces the parallel composition $P_1|P_2$ which represents two different processes running concurrently and the scope restriction where we say that a and b are bound names in P.

We now show the reduction rules to writing and receiving processes.

$$(\nu xy)(x!vP \mid y?z.Q) \rightarrow (\nu xy)(P \mid Q[v/z])$$
$$(\nu xy)(x!vP \mid *y?z.Q) \rightarrow (\nu xy)(P \mid Q[v/z] \mid y?*z.Q)$$

In the first case channel creation binds two new variables, called x and y, that may be used in processes composed of the output of value v on channel x and a process ready to receive this value and to replace z by v in the continuation process. The prefixes are then consumed but the scope restriction remains in the resulting process in order to be used in processes P and Q. The input of the second case is replicated. The reduction rule is similar except that the replicated input remains in the resulting process.

2.2 Session types

Session types are a formalism that allows a concise description of the continuous interactions among different partners in a concurrent computation [35, 36].

Session types were first introduced with the purpose of specifying interactions between two participants running in parallel and communicating via message passing [21, 31]. These two works propose an extension of the pi-calculus with session types allowing to specify structured patterns of communication and verify whether processes are well-formed via type checking. Later, Gay and Hole [16], introduced a notion of subtyping for session types, while working on a more conventional pi-calculus.

Consider two processes that can communicate through a channel defined by two end-points, named x and y. In order to write an integer on channel x and to read on channel y, the session types that describe the behaviour of x and y are, respectively, $!\mathsf{integer}.\mathsf{end}$ and $?\mathsf{integer}.\mathsf{end}$, where the output is represented by $!$ and the input by $?$. The end type means that no further interaction may occur in the given channel end. On the other hand, if the objective is to offer a set of options on channel end x and to select on channel y, the types of the end-points x and y are $\&\{l_i\colon T_i\}_{i\in I}$ and $\oplus\{l_i\colon T_i\}_{i\in I}$, respectively, where branching is represented by $\&$, selection by $\oplus$ and I is an index set.

Session types are used to describe interactions between exactly two threads. Certain communication patterns require channels shared by more than two threads. In order to describe linear and shared objects, and based on the ideas of a linear type system of functional programming [37], we may equip a pretype with a *linear* qualifier to obtain a traditional session type, or with an *unrestricted* qualifier to describe a channel shared by an unbounded number of threads.

Baltazar et al. combine session types with linear refinements resulting in an original system of *linearly refined session types* [3]. *Refinement types* are a form of dependent types which allow attaching formulae to types [14], thus specifying properties of values in programs. The refinement type $\{x\colon T \mid A\}$ represents a value v of type T that must respect formula A. Formula A may refer to v via variable x. For instance, the type $\{x\colon \mathsf{integer} \mid x \geqslant 0\}$ describes a natural number.

Honda et al. presented the Scribble framework [18, 19], whose purpose is to provide a formal and intuitive language and tools to specify communication protocols and their implementations, using the theory of multiparty session types. Multiparty session types are described by Honda et al. [22] as an extension of binary session types able to describe interactions among multiple partners of the communication.

A *protocol* describes globally the interactions among two or more participants and stipulate, for each participant, its *role* in the communication. Scribble includes constructs to describe interactions in protocols, such as, the *interaction signature* to specify what kind of message is sent from one participant to another one, the *sequencing* to represent a sequence of multiple interactions where a given role name appears, the *parallel* or *unordered* to represent interactions that may occur in any order, the *directed choice* to define an interaction in form of branching, the *recursion* to repeat an interaction and, finally, the *nested protocol* where a new interaction may be instantiated in a nested protocol. Using the Scribble's development tools, a programmer may specify a set of protocols to be used in a given program, check if they are valid and free from deadlocks and he may validate a program against these protocols using a *protocol type checker*. Scribble supports bindings for various high-level languages such as ML, Java, Python, C# or C++.

2.3 Session types in functional languages

In this section we describe how the notion of session types was introduced in functional programming languages.

Neubauer and Thiemann presented an implementation of session types in Haskell [27]. In this language, communicating parties exchange messages via channels whose behaviour is described by session types encoded in terms of type classes with functional dependencies, using the *session monad*. To model the parties (for instance a server or a client) the authors use functions with polymorphic parameters. The operations supported are recursive types, message sending and reception (each message is a value tagged with a label) and the closing of a communication channel, which requires that the closed channel has reached its end meaning that there are no message exchanges left.

Sackman and Eisenbach describe a similar work that also incorporates session types in Haskell [30]. In this implementation, the communication is made via bidirectional channels. The behaviour of each channel is governed by a session type, in order to ensure

that the two processes that use this channel communicate smoothly. In order to construct session types, the authors use a Domain Specific Language (DSL), which works within an extended *Monad* type class (each channel is represented by a value which is an instance of this type class). The use of DSL allows to assign labels to session types, or fragments of session types, and to refer them using these labels. This language also supports operations to send and receive messages, and in addition, it includes primitives to offer, select, jump and termination (end). The send and receive operations are used to exchange messages, the choice operations have a behaviour similar to a switch statement, where a process produces a list of options (offer) and another one chooses one option from this list (selection). The jump operation is used to specify recursive types and loops (including infinite loops).

Both papers prove that session types can be embedded in Haskell in a type-safe way and that it is possible to encode all the invariants and properties of session types in the Haskell type system, allowing to statically verify the use of communication primitives without any modification to the compiler, type checker or preprocessor.

Given that session types are encoded, the Haskell code can be difficult to read. SePi works directly with session types, thus hopefully leading to readable programs.

Bhargavan et al. [4] present a high-level language to specify multiparty sessions. Their compiler generates from a session language cryptographic protocols encoded as ML modules and proof annotations, making sure that messages are exchanged with strong security guarantees such as integrity and secrecy. The verification of security of the generated code is based on the work of Bhargavan et al. [5] that verifies *executable protocol code* instead of abstract protocol models. Participants of a session are represented by *roles* where each role has its local implementation and code for *sending* and *receiving* messages (the compiler generates cryptographic operations to this code). The patterns of communication allowed between two roles are defined by session types. In addition to the send and receive operations, the language also provides for *loops* and *branches*.

2.4 Session types in object-oriented languages

In this section we introduce a few object-oriented languages equipped with session types.

Fähndrich et al. present a type-safe object-oriented and garbage collected programming language called Sing#, a variant of C#, that supports message-based communication via shared-memory [12]. This language was used to write the Singularity operating system [24], ensuring process isolation (a process cannot access or corrupt data or code of another process) and inter-process communication (processes may exchange messages and signal events), two important services of the operating system. Communication in Sing# is via bidirectional channels, where each channel is characterized by two end-points, one to read and the other to write, that are created by a specific *channel creation* operation. Channels can be used to receive, to asynchronously send messages and also to offer a set

of options, using a *switch-receive* statement. Their behaviour is governed by statically verified contracts, a mechanism similar to session types.

Hu et al. introduced SJ, an extension of Java with a concise and clear syntax for session types and structured communication operations [23]. SJ is a language for session-based distributed programming that features asynchronous message passing, delegation, session subtyping and interleaving. SJ ensures communication safety for distributed applications via a combination between static and dynamic validations (the first to ensure that each session behaves as prescribed by a locally declared protocol and the second to verify whether the parties of the communication implement compatible protocols). It also supports session abstraction over concrete transports, that is, session operations are mapped to runtime communication primitives that can be implemented over concrete transports, using TCP. SJ programming can be divided in the definition of communication protocols (session types) and in their implementation using the session operations. This implementation requires the creation of *session sockets*. A client may use a session socket (representing one end-point of the communication) to request a new session to the server using a *session server-address* (this one is responsible to define the address of a server using its IP address and a TCP port). When a *session server socket* accepts the request, it creates a new session socket (the other end-point), to be used in the server side to communicate with the client. After the session is established, session sockets may be used to send and receive messages, iterate or offer a menu of options.

Bica and Mool are two object-oriented programming languages where session types describe the order by which methods in classes should be called. Caldeira and Vasconcelos [8] presented an extension to Java5, called Bica, that checks Java source code against session types specifications for classes , based on the work by Gay et al. [15]. The extension allows attaching session types to classes, in the form of Java annotations, that specify the possible orders of method calls, as well as the tests that clients must perform on the results of method calls.

Following a similar approach, Campos and Vasconcelos [9, 10] introduced the Mool programming language, a mini object-oriented language based on Java, with support for concurrency. Mool formalizes protocols, called *usage types*, that define how and when the method of a class should be called. These protocols are attached to class definitions in order to specify the available methods, what tests clients must perform, and the object status: linear or shared. While in SePi and in general approaches the communication occurs by exchanging messages on session governed channels, in this language communication occurs by calling methods on session governed object references, as described in [35].

2.5 Session types in imperative languages

Ng et al. presented a programming framework for message-passing parallel algorithms which combines session types with the C programming language, called Session C [28]. This multiparty session-based programming environment ensures deadlock freedom, communication safety and global progress for well-typed programs. A Session C program is a C program that uses communication primitives based on the theory of session types. Besides the usual operations to send and receive values, Session C also includes multicast sending and multicast receiving primitives, where the first one sends the same message to all receivers and the second one receives messages from multiples senders. It also provides a branching operation where a programmer may define different communication behaviours (different options) for a participant according to what option is selected by another participant. Finally, this implementation also provides two methods for *iteration*: local and communicating. A *local iteration* is similar to a *while statement* and a *communicating iteration* is a distributed version of a loop to support multicast.

The Session C framework uses the programming language Scribble to describe communication protocols in the form of multiparty session types.

2.6 Programming languages based on the pi-calculus

Pierce and Turner present the programming language Pict [29], a strongly typed concurrent programming language in the ML-tradition, directly based on pi-calculus, and equipped with a combination of subtyping and polymorphism. Pict builds on a tiny core (a variant of the asynchronous pi-calculus [7, 20]) a few derived constructs. The core language contains two kinds of entities: processes (also called agents) and channels (or names), where processes use channels to communicate with other processes. Processes include asynchronous output, input prefix, parallel composition, replicated input, conditional and local declaration (creation of a new channel with a type associated to the created channel). The language grows with some derived constructs such as multiple declarations, a primitive *run* operation, to run a process in parallel with the rest of the declarations, a process abstraction in the form of a declaration using the keyword *def*, which can be translated to a channel creation followed by a replicated input and mutually recursive definitions. Pict supports types to describe which values are sent or received through communication channels, recursive types and subtyping.

Vasconcelos presented TyCO, an object-based concurrent programming language, which uses a variant of the asynchronous pi-calculus to capture the notions of concurrent objects [33, 34]. TyCO programs are composed of asynchronous labelled messages (atomic select/output) and labelled receptors (branch/input), concurrent objects composed of labelled methods, concurrent composition and an operator to create a new channel. TyCO uses the notion of *agents* to represent processes abstracted on a set of channels

allowing for recursion (*recursively abstracted agents*) and for using more than once a declared agent (*simply abstracted processes*) via the let constructor. Vasconcelos also added to this language a few derived constructs such as *datatype declarations*, *constructed data*, *case expression*, similar to our branching construct which can be used to write a conditional expression, or *functional objects*.

Although typed on pi-calculus, neither of these languages uses session types. These languages, and all others presented in this chapter do not include refinement types, linear or classic.

Chapter 3

The SePi programming language

This chapter presents the SePi programming language, its type system and operational semantics. Section 3.1 presents the syntax of our core language, describing processes, types and formulae. Section 3.2 introduces the final SePi language, containing abbreviations, new constructs derived from the core language, new primitive values and expressions. Section 3.3 presents the algorithmic rules to type check the core language. Section 3.4 discusses the derived constructs in the language. Finally, Section 3.5 presents programming examples attesting the flexibility of SePi.

3.1 The core language

The syntax, type system and operational semantics of our core language are introduced in references [3, 36]. Figure 3.1 shows the syntax of processes and values in the core language.

Processes require two base sets, that of program variables, ranged over by x, y, ... and that of choice labels ranged over by l, m, ... The channel creation construct binds two new variables in P, one for each of the channel's end-points. One of these variables is used in process P to write values into the channel while the other is used to read. Variable x is of type T, whereas variable y is of type dual of T (a notion discussed below). The output process $x!v.P$ is used to send (or to write) value v on channel end-point x before continuing with the process P. The input process $x?y.P$ reads on channel end-point x a value, before continuing with process P, where the received value replaces the bound variable y. Input processes are *linear*, $x?y.P$, or *replicated*, $x*?y.P$. The difference between them is that the second remains after the reception of the value while the first does not, meaning that a replicated input can be used by multiple clients whereas a linear input can be used by a single client. In this way, a replicated input provides for unbounded behaviour. The parallel composition $\{P_1 \mid ... \mid P_n\}$ allows n processes to run concurrently. A parallel composition of zero processes, $\{\}$, represents the terminated process. The conditional construct $\textsf{if}\ v\ \textsf{then}\ P\ \textsf{else}\ Q$ executes P if the boolean value v is

P ::=		Processes:
	new $xy\colon T\ P$	channel creation
	$x!v.P$	output
	$x?x.P$	linear input
	$x{*}?x.P$	replicated input
	$\{P_1 \mid ... \mid P_n\}$	parallel composition, $n \geq 0$
	if v then P else P	conditional
	x select $l.P$	selection
	case x of $l_1 \rightarrow P_1 \ \cdots l_n \rightarrow P_n$	branching, $n \geq 0$
	assume A	assume
	assert $A.P$	assert
v ::=		Values:
	x	variable
	true | false	boolean values

Figure 3.1: Syntax of core processes and values

true, and Q otherwise. There are two choice constructs: the selection process x select $l.P$ denotes a process, that chooses, on channel end x, the option labelled with l from a set of multiple options, before continuing as process P. This set of options is offered by a branching process of the form case x of $l_1 \rightarrow P_1 \ \cdots \ l_n \rightarrow P_n$. The construct assume A denotes a process that introduces an assumption A in the form of a formula. Conversely the assertion process, assert $A.P$, checks that formula A holds before continuing with P. Assumptions and assertions are treated *linearly*: for each **assert** there must be exactly one **assume** and, conversely, for each **assume** there must be exactly one **assert**.

Values include the boolean literals, true and false, as well as variables denoting channel ends.

The syntax of types is introduced in Figure 3.2. Types rely on one further base set, that of type variables, ranged over by a, b, ... Types include the primitive type boolean, used to describe boolean values, the termination type end, used to describe end-points where no further interaction is possible, qualified pretypes, type variables, recursive types and refinement types.

Qualified pretypes represent channel end-points ready to

- $!x\colon T.U$, send a value of type T and then continuing its interaction as defined by type U, where the value sent replaces (free) occurrences of x in U.

- $?x\colon T.U$, receive a value of type T before behaving as defined by type U, where the value received replaces (free) occurrences of x in U.

q ::=	Qualifiers:
lin	linear
un	unrestricted
p ::=	Pretypes:
$x\colon ?T.T$	receive
$x\colon !T.T$	send
$+\{l_i\colon T_i\}_{i\in I}$	select
$\&\{l_i\colon T_i\}_{i\in I}$	branch
T ::=	Types:
$\mathsf{boolean}$	boolean
end	termination
$q\,p$	qualified pretype
a	type variable
$\mathsf{rec}\ a.T$	recursive type
$\{x : T \mid A\}$	refinement

Figure 3.2: The syntax of types

- $+\{l_i\colon T_i\}_{i\in I}$, select an option labelled with one of the labels in set $\{l_i\}_{i\in I}$, before behaving as T_j if the label l_j is selected, for some index set I.
- $\&\{l_i\colon T_i\}_{i\in I}$, offer a set of options, each labelled with a different l_i, and behaving as prescribed in T_j if label l_j is selected.

Qualified pretypes qp describe how many parties (or threads) know the communication medium: linearly qualified pretypes ($q = \mathsf{lin}$) represent channel end-points that occur in exactly one thread and unrestricted pretypes ($q = \mathsf{un}$) represent channel ends that may appear in an unbounded number of threads.

The recursive type rec $a.\,T$ represents a channel end that behaves according the type T with all occurrences of a replaced by rec $a.\,T$. For instance, the type rec $a.$ lin!boolean. lin?boolean.a defines an end-point that after sending and receiving a boolean value is ready to send and receive boolean values again. These types are required to be *contractive*, that is, they may not contain subexpressions of the form rec $a_1 \dots$ rec $a_n.a_1$, for $n \geqslant 1$.

Finally, the refinement type constructor, $\{x\colon T \mid A\}$, is used to incorporate logical information into session types. Such type describes a channel end of type T that conforms to formula A. Formula A may refer to variable x or to data appearing "previously" in the type via the bound type variables in send and receive types. For instance, the type

A ::=	Formulae:
$p(v_1, ..., v_n)$	predicate on v_1, ..., v_n
$A * A$	joining
unit	identity

Figure 3.3: Syntax of formulae

$\text{lin}?x\colon \text{integer}.\text{lin}?\{y\colon \text{integer} \mid y > x\}$ describes a channel end that receives an integer and then receives another integer greater than the first one.

Figure 3.3 shows the syntax of formulae. There are three kinds of formulae: the uninterpreted predicate constructor, which may refer to channel names or primitive values (boolean values in case of our core language), the linear logic connective of tensor, $*$, and the identity constructor, unit.

Program variables may occur in processes, types and formulae, whereas type variables may occur in types only. We say that program variable y occurs *bound* in Q in processes of the form $qx?y.Q$ or new $yz\colon T\ Q$. In the latter case, z is also bound in Q. We also say that the same program variable is bound in U in types of the form $q?y\colon T.U$ and $q!y\colon T.U$, and in formulae A in types of the form $\{y\colon T \mid A\}$. A variable that is not bound is said to be *free*. The set of free variables in a process, type or formula is denoted by $\text{free}(P)$, $\text{free}(T)$, $\text{free}(A)$, respectively. We omit the inductive definition. For type variables we say that type variable a occurs bound in U in types of the form rec $a.U$.

3.2 The SePi language

3.2.1 Rationale

The SePi language is based on the core language introduced above. However it includes further primitive types, expressions that extend values, abbreviations and a few derived constructs, such as process definitions, type declarations and input/output of multiple values.

We can write multiple programs with the core language alone. In fact, except for the new primitive types and expressions, every program that we write in SePi can be written in the core language, since the new constructs are derived from those in the core language. The SePi language however allows us to capture, with a special syntax, the most common idioms and programming patterns, thus reducing the number of lines of code produced and the potentially associated errors.

The SePi language further allows arithmetic expressions (+, binary and unary −, %, / and *), logical expressions (**and**, **or** and **not**) and relational expressions (<, >, <=, >=, ==, /=).

3.2.2 Introducing the language via an example

We use a running example to informally introduce the SePi language. The example is that of an *online donation service*, that manages donation campaigns, based on the *online petition service* [35] and on the *online store* [3] examples.

Clients seeking to start a donation campaign for a given cause begin by setting up a session with the server. The server should create a new channel and respond to the client, sending an end-point of the new channel. The session is conducted by this channel on which the campaign related data is provided (title and deadline for donation collection for example). The server should offer, a menu of options to edit the campaign data. When a client finishes the campaign creation, the promotion phase starts. During this phase, the campaign channel may be disseminated and used by different donors for the purpose of collecting donations. Parties donating for some cause do so by providing a name, a credit card number and an amount to be charged in the card. When the server receives the donation data it forwards these information to the bank.

We may divide the example in three distinct parts: the bank, the donation server and the clients. Communication happens between the bank and the server, and between the server and the clients, which means that clients never communicate directly with the bank. Communication among the different participants is by message passing on bidirectional synchronous channels.

We start with a few type abbreviations that will ease programming.

```
1  type CreditCard = string
2  type Promotion = *!(string, c: CreditCard, {x: integer | charge(c, x)})
3  type Decision = &{accepted: Promotion,
4                    denied: ?string.end}
5  type Donation = +{setTitle: !string.Donation,
6                    setDate: !integer.Donation,
7                    submit: Decision}
```

The first lines of our program (lines 1–7) show some examples of type abbreviations. The type declaration construct, **type** a = T, introduces the name T representing the solution of equation a = T (details in Section 3.4). Line 1 says that CreditCard is another name to type **string**. On line 2, we have a prefix type, !, that represents an unbounded number, $\star$, (in sequence or in parallel) of outputs of multiple values each. In other words, we may use a channel with type Promotion to send, as many times as needed, a triple composed of a string, a value of type CreditCard (another string) and an integer value that respects the formula charge(c, x), where c denotes the credit card and x the amount to be charged. Input and output of multiple values are natural extensions of those in Figure 3.2. The details are discussed in Section 3.4. The $\star$ syntax introduces an unrestricted recursive type. In general, $\star$!T abbreviates **rec** a. **un**!T.a where a does not occur in T.

In the Decision type we have a branching choice type (&). A channel with this type should be used in a **case** process with labels accepted and denied. In type Donation we

```
dualof q!(x1:T1, ..., xn:Tn).U = q?(x1:T1, ..., xn:Tn).dualof U
dualof q?(x1:T1, ..., xn:Tn).U = q!(x1:T1, ..., xn:Tn).dualof U
dualof q&{l1:T1; ...; ln:Tn} = q+{l1:dualof T1; ...; ln:dualof Tn}
dualof q+{l1:T1; ...; ln:Tn} = q&{l1:dualof T1; ...; ln:dualof Tn}
dualof rec a.T = rec a.dualof T
dualof a = a
dualof end = end
```

Figure 3.4: The **dualof** partial function

have a selection choice (+); in this case, the channel should be used on a **select** process, with one of the three labels, setTitle, setDate, or submit.

We also add to the SePi language another abbreviation that it is not used in this example. The unrestricted choice types **rec** a. **un**&{l: a; m: a} and **rec** a. **un**+{l: a; m: a} may be abbreviated to *&{l; m} and *+{l; m}, respectively. These abbreviations allow to succinctly describe the types of (shared) boolean values: *&{True; False} and *+{True; False}.

Prefix and choice types are qualified as linear (**lin**) or unrestricted (**un**). In our example we all omit qualifiers since the **lin** keyword is optional and we may use * to represent the most common class of unrestricted types, as in Promotion. As a result, explicit occurrences of the **lin**/ **un** qualifiers are rarely needed in SePi.

The Promotion type shows an example of refinement types. Refinement types describe properties of exchanged values. The type {x: **integer** | charge(c, x)} describes an integer value for which the charge(c, x) capability must be respected.

There is an important concept in our language—*duality*. The two end-points of a channel are supposed to have a dual behaviour. That is, when an end-point sends a value, the other must be ready to receive it and, in the same way, when an end-point offers a set of options, the other must be ready to select one of the multiple option. Figure 3.4 shows the definition of the dual function. Duality is not defined for **boolean**, **integer**, **string** and refinement types. We introduce, in the syntax of our language, an abbreviation to obtain a dual type, **dualof**, allowing programmer to obtain the dual type of a session type.

Now we introduce the clients of our example. Each client receives from server a channel end of type Donation that allows to create a new campaign. Clients create a new donation campaign by choosing its name and the deadline for the donative collection. These two operations can be performed in any order and more than once each, due the recursion present in the Donation type when the setDate and setTitle options are chosen. When a client is happy with the donation data, he submits the proposal and waits for the server's reply. The server evaluates the received data and decides whether the campaign should be accepted or not. If accepted, the client may start promoting the donation, otherwise, when denied, the client receives a string with the reason. During the promotion phase, a client may donate and/or disseminate the donation channel.

```
8  // A client in two parts
9  def helpSavingTheWolf (ps: *?Donation) =
10     def donate(p: Promotion, donor: string, ccard: CreditCard, amount:
           integer) = {
11         assume charge(ccard, amount) |
12         p!(donor, ccard, amount)
13     }
14     ps?p.
15     p select setDate. p!2012.
16     p select setTitle. p!"Help Saving the Wolf".
17     p select setDate.p!2013.
18     p select commit.
19     case p of
20         accepted → {
21             Donate!(p, "Donor1", "2345", 5) |
22             Donate!(p, "Donor2", "1234", 10) |
23             Donate!(p, "Donor3", "1004", 20)
24         }
25         denied → p?reason. printString!reason
```

An example of a client is in lines 8–25. The client code is divided in two parts: the HelpSavingTheWolf and the Donate processes. The first definition shows some examples of the constructs described in Section 3.1, such as input (line 14) and output (line 25) of one value, selection (line 15), branching (lines 19–25) and assume (line 11), but it also shows new constructs, such as the process definition (**def**) and the output of multiple values (line 21). The client uses selection followed by output (in lines 15–18) to choose the name and the date of the campaign, a selection process to commit the data, and the branching process of lines 19–25 to offer the server the accepted and denied options, meaning that he is waiting for an answer. If the server accepts the donation the execution continues on lines 21–23, otherwise, it continues on line 25.

The process definition construct, represented by the keyword **def**, is an abbreviation of a channel creation with a replicated input in parallel with the rest of the program. For instance, the meaning of the **def** Donate is given by the following process:

```
new donate donateReader: *!(Promotion, string, CreditCard, integer)
    donateReader*?(p, donor, ccard, integer).
    assume charge(ccard, amount) |
    p!(donor, ccard, amount)
```

Given that **def** is an abbreviation for an input, we invoke definitions using conventional output processes, as in line 17, donate!(p, "Donor1", "2345", 5).

Our core language is based on the monadic pi-calculus [26]. However it includes constructs to send and receive multiple values, free from interference. Such constants derive from output and input of core language, as explained by Milner [25] and Vasconcelos [36]. The details can be found in Section 3.4. One example of output of multiple values is in line 21 where it is used to invoke the process definition donate (defined in lines 10–13): in lines 21–23 there are three clients donating to the campaign helpSavingTheWolf.

The channel end p, which behaves as defined by Donation type, is linear while the

client is sending the donation data but becomes unrestricted when the server accepts the campaign. Due the linear behaviour of the channel during the setup phase we are sure that there are no *race conditions* while reading/writing from the channel, whereas, due its unrestricted nature, during the promotion phase, multiple clients may concurrently try to donate.

In order to make possible printing primitive values we add to SePi language three channel end-points: printBoolean, printInteger and printString. These channel ends must be used in output processes, according to their types: *!**boolean**, *!**string** and *!**integer**. See an example in line 25.

Figure 3.1 shows that input, output and selection processes have a continuation process, but as we can see in our code, for instance in line 21, the process printString!reason does not present any continuation. This happens because continuations of the form {} may be omitted.

The donate definition receives as parameters a channel with type Promotion, two strings with the name of the donor and the credit card and an integer with the donated amount. In our example, a donation consists in sending to server the donor name, his credit card number and the amount to be donated. But type Promotion (line 2) describes a channel that sends a triple with a string, a credit card and a refinement type describing an integer that respects the formula charge(c, x), meaning that a simple output process of the form p!(donor, ccard, amount) it is not enough. Due the refinement type, the client must **assume** first the formula charge(ccard, amount), as in line 11, which will eventually allow the bank to charge the credit card.

We now explain the server side code. Our server is divided in one main process definition, the donationServer, and four auxiliary process definitions: the promotion, the denied, the accepted and the setup.

```
26 // The donation server in five parts
27 def donationServer (ps: *!Donation) =
28     def setup (p: dualof Donation, title: string, date: integer) =
29         case p of
30             setDate  → p?d. setup!(p, title, d)
31             setTitle → p?t. setup!(p, t, date)
32             commit   → if date < 2013 then denied!p else accepted!p
33     def denied (p: dualof Decision) =
34         p select denied.
35         p!"We can only accept 2013 donations\n"
36     def accepted (p: dualof Decision) =
37         p select accepted.
38         promotion!p
39     def promotion (p: dualof Promotion) =
40         p?(donor, ccard, amount).
41         promotion!p.    // recur
42         bank!(ccard, amount)    // charge the credit card
43
44     new p1 p2: Donation
45         ps!p1.
```

```
46        setup!(p2, "Help me", 2000).    // call with default values
47        donationServer!ps    // recur
```

The donationServer starts with the creation of a new channel, in line 44. The keyword **new** creates a new channel defined by the two ends p1 and p2 and allows the ensuing process to use them. In lines 44–46, we have an example of the technique known as *session initiation* where the server creates a new channel, sends the end-point p1 to the client, and keeps the other, p2, to itself, in order to receive the requests from client. The rest of the code should be easy to follow based on the preceding explanation.

The behaviour of the setup process (lines 28–32), is that of a loop. It collects data from client and when the client commits, it evaluates whether the campaign should be accepted or not, using a conditional process. If accepted then the server delegates the channel p to the accepted process, otherwise to the denied process. The denied process selects on channel p the option denied and sends to client the reason for denial option. The accepted process selects the accepted option and the protocol passes to the promotion phase, where the server waits for donations.

The promotion process receives the donor name, the credit card and the defined amount and sends them to the bank, by invoking the bank process.

Lines 48–50 present a simplistic bank that receives a credit card and the amount to be charged and that requires the capability charge(ccard,amount) to have been granted.

```
47  // The bank that charges credit cards
48  def Bank (ccard: CreditCard, amount: {x: integer | charge(ccard, x)}) =
49      assert charge(ccard, amount).
50      printInteger!amount
```

When a client donates an amount, he wants to be sure that his credit card is used exactly once, and charged the agreed amount (not more, not less). In order to avoid misbehaving donation servers that forward an incorrect amount or that invoke the Bank definition twice we use the refinement type {x: **integer** | charge(ccard, x)} together with **assume** and **assert** processes. First, in the donate definition, the client assumes charge(ccard, x) and then the bank asserts the same formula (line 49). But the bank can only assert this formula if, during the promotion phase, the server forwards the exact values received from the client.

However, there is one way to charge a credit card twice or more: assuming on behalf of the client and asserting on behalf of the bank. For instance if we replace the current promotion process by the following one:

```
39  def promotion (p: dualof Promotion) =
40      p?(donor, ccard, amount).
41      promotion!p.    // recur
42      assert charge(ccard, amount).{
43          assume charge(ccard, amount)*charge(ccard, amount) |
44          bank!(ccard, amount).bank!(ccard, amount)
45      }
```

Finally, lines 52–54 show the main process of our program. It consists the creation of a new channel defined by two end-points ps1 and ps2 where the first is sent to the donationServer and the second to the helpSavingTheWolf client, thus allowing communication between both.

```
51 // Main
52 new ps1 ps2: *!Donation
53     DonationServer!ps1 |
54     HelpSavingTheWolf!ps2
```

3.3 Algorithmic type checking

To complete the description of the language we introduce the static semantics (type checking) and the operational semantics of the language. Table 3.1 summarises where the rules can be found.

		Rules		
		Declarative system	**Algorithmic system**	**Implementation detais**
Type checking	**Session types**	Fundamentals of Session Types [36]	Fundamentals of Session Types [36]	This thesis (Chapter 4)
	Refinements	Linearly Refined Session Types [3]	This thesis (Section 3.3)	This thesis (Chapter 4)
Reduction		Fundamentals of Session Types [36]	The Polymorphic Pi-calculus: Theory and Implementation [32]	This thesis (Chapter 4)

Table 3.1: Provenance of the rules for type checking and reduction

Processes, types and formulae are checked against *typing contexts*. A typing context Γ is a finite map from variables to types. A context can be empty, $\cdot$; of the form $\Gamma, x\colon T$ to represent a map which contains the entry $x\colon T$; or of the form Γ, A to represent a map that contains a formula A. Notation $\mathsf{dom}(\Gamma)$ denotes the set of variables associated to type entries: $x \in \mathsf{dom}(\Gamma)$ if $x\colon T \in \Gamma$.

There are two important typing contexts operations. The *context update* $+$ which, given a variable x and a type T returns $\Gamma, x\colon T$ if there is no entry for x in Γ, or returns Γ if $x\colon T$ is already in Γ and T is unrestricted. The *context difference* (or quotient) $\div$, which given a typing context Γ and a set of program variables L removes from the context the entries associated to the variables in L *if* their types are unrestricted, and is undefined if

the type is linear. Note that a type is unrestricted when is pretype qualified as unrestricted, boolean, integer, string or end. The formal definition is in [36].

There is another important concept in our language: the *substitution* of variable x by value v in a type or a formula, denoted by $[v/x]T$ and $[v/x]A$, respectively. For instance the substitution $[v/x]\{x\colon \mathsf{boolean} \mid p(x)\}$ results in type $\{x\colon \mathsf{boolean} \mid p(v)\}$. One cannot replace variables by expressions.

3.3.1 Well-formed types and formulae

Types and formulae may contain free program variables, but these must be declared in the typing context. Sets $\mathrm{free}(T)$ and $\mathrm{free}(A)$, the free program variables in T and A were introduced in Section 3.1.

An example of a well-formed formula with respect to context $x\colon \mathsf{boolean}, y\colon \mathsf{integer}$ is $p(x)$ because the set of free variables of $p(x)$ is $\{x\}$, $\mathsf{dom}(\Gamma) = \{x, y\}$ and $\{x\} \subseteq \{x, y\}$. Such variables must have been declared, which in our case means to be in the domain of the typing context. We write $\Gamma \vdash T$ when the free variables of T are in the domain of Γ, and similarly for formulae.

$$\frac{\mathsf{free}(T) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash T} \qquad \frac{\mathsf{free}(A) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash A}$$

3.3.2 Formulae normalisation, $\text{normalise } A \mapsto \Gamma$

All rules below are the form of $X \mapsto Y$ where X represents the input and Y the output of the rule.

The purpose of formulae normalisation is to obtain a multiset of the uninterpreted predicates that the formula contains, which turns out to be a typing context. The normalisation of the unit formula returns the empty list, the normalisation of a predicate returns the predicate itself and the normalization of a tensor invokes the normalisation of the left formula and the right formula.

$$\begin{aligned}
\text{normalise } \mathsf{unit} &\mapsto \cdot \\
\text{normalise } p(v_1, \cdots, v_n) &\mapsto \{p(v_1, \cdots, v_n)\} \\
\text{normalise } A_1 * A_2 &\mapsto \text{normalise } A_1, \text{normalise } A_2
\end{aligned}$$

3.3.3 Context normalisation, $\text{normalise } \Gamma \mapsto \Gamma$

The purpose of context normalisation is to extract all formulae from refinement types. The normalisation of an empty context results in an empty context. If the context contains a non refinement type then this type remains and the rest of the context is normalised. Otherwise, if the context contains an entry with refinement type in form of $x\colon \{y\colon T|A\}$

then we further normalise $x\colon T$ (for may be a refinement type). We must also normalise formula A after replacing all occurrences of variable y for x.

$$
\begin{aligned}
\text{normalise } \cdot &\mapsto \cdot \\
\text{normalise } (\Gamma, x\colon T) &\mapsto \text{normalise } (\Gamma), x\colon T \quad \text{if } T \neq \{y\colon U|A\} \\
\text{normalise } (\Gamma, x\colon \{y\colon T \mid A\}) &\mapsto \text{normalise } (\Gamma, x\colon T), \text{normalise } [x/y]A
\end{aligned}
$$

3.3.4 Formula subtraction, $\Gamma \vdash A \mapsto \Gamma$

The purpose of formula subtraction is to remove from the multiset of predicates a given formula.

There are three kinds of formulae: the unit, the uninterpreted predicate and the tensor formula. Subtracting the unit formula from a context results in the original context. Subtracting a predicate $p(\vec{v})$ results in a context without $p(\vec{v})$. Finally, subtracting a tensor formula $A_1 * A_2$ results in the subtraction of formulas A_1 and A_2 individually.

$$\Gamma \vdash \mathsf{unit} \mapsto \Gamma$$

$$\Gamma_1, p(\vec{v}), \Gamma_2 \vdash p(\vec{v}) \mapsto \Gamma_1, \Gamma_2$$

$$\frac{\Gamma_1 \vdash A_1 \mapsto \Gamma_2 \qquad \Gamma_2 \vdash A_2 \mapsto \Gamma_3}{\Gamma_1 \vdash A_1 * A_2 \mapsto \Gamma_3}$$

3.3.5 Type equivalence, $\Gamma \vdash v\colon T \equiv T \mapsto \Gamma$

The type equivalence rules compare the infinite trees associated to types. The co-inductive definition is outside the scope of this thesis. We however show how type equivalence deals with refinement types, using an inductive version of a rule that compares a refinement type of form $\{x\colon T_1|A\}$ with an arbitrary type T_2. The rule first compares the inner type of refinement, T_1, with the second type T_2 and subtracts the formula A after replace all occurrences of x by v.

$$\frac{\Gamma_1 \vdash v\colon T_1 \equiv T_2 \mapsto \Gamma_2 \qquad \Gamma_2 \vdash [v/x]A \mapsto \Gamma_3}{\Gamma_1 \vdash v\colon \{x\colon T_1|A\} \equiv T_2 \mapsto \Gamma_3}$$

3.3.6 Typing rules for expressions, $\Gamma \vdash e \mapsto T; \Gamma$

A sequent of the form of $\Gamma_1 \vdash e \mapsto T; \Gamma_2$ assigns type T to expression e, given a context Γ_1, and producing a new context Γ_2. Γ_2 may differ from Γ_1 due the behaviour of linear

channels.

$$\Gamma \vdash \mathsf{true} \mapsto \mathsf{boolean}; \Gamma \qquad \Gamma \vdash \mathsf{false} \mapsto \mathsf{boolean}; \Gamma \qquad \text{[A-TRUE] [A-FALSE]}$$

$$\frac{\Gamma_1 \vdash e_1 \mapsto \mathsf{integer}; \Gamma_2 \qquad \Gamma_2 \vdash e_2 \mapsto \mathsf{integer}; \Gamma_3}{\Gamma_1 \vdash e_1 + e_2 \mapsto \mathsf{integer}; \Gamma_3} \qquad \text{[A-PLUS]}$$

$$\frac{\mathsf{un}(T)}{\Gamma_1, x\colon T, \Gamma_2 \vdash x \mapsto T; (\Gamma_1, x\colon T, \Gamma_2)} \qquad \text{[A-UNVAR]}$$

$$\Gamma_1, x\colon \mathsf{lin}\, p, \Gamma_2 \vdash x \mapsto \mathsf{lin}\, p; (\Gamma_1, \Gamma_2) \qquad \text{[A-LINVAR]}$$

Rules [A-TRUE] and [A-FALSE] say that boolean values have type boolean and, similarly, integer and string values have types integer and string (not shown). We also present an example of algorithmic rule for a binary expression, the addition. The rule returns the integer type if both expressions have integer types. The incoming context Γ_1 is passed to the call for e_1 obtaining context Γ_2 that is passed to the call for e_2. The output of the second call is the output of the binary expression. We can easily see that, in this case, $\Gamma_1 = \Gamma_2 = \Gamma_3$. Rules [A-UNVAR] and [A-LINVAR] return, for a variable x, the type contained in the context. The first rule keeps the entry $x\colon T$ in the context, if the resulting type T is unrestricted, while the second removes the entry if the type is linear.

Predicate un (T) says that type T is unrestricted and is true if T is boolean, integer, string, end or if it is a pretype classified as unrestricted.

3.3.7 Typing rules for processes, $\Gamma \vdash P \mapsto \Gamma; L$

The process type checking function as input receives a typing context Γ and a process P, and returns a new context Γ_2 and a set L of variables. L contains the free linear variables in P that occur in a *subject* position. We say that x occurs bound in subject position in the following processes: $x!e.P$, $x?y.P$, x select $l.P$ and case x of $l_i \to P_i$.

$$\frac{\Gamma_1 \vdash P_1 \mapsto \Gamma_2; L_1 \quad \Gamma_2 \div L_1 \vdash P_2 \mapsto \Gamma_3; L_2 \quad \dots \quad \Gamma_n \div L_{n-1} \vdash P_n \mapsto \Gamma_{n+1}; L_n}{\Gamma_1 \vdash \{P_1 \mid \dots \mid P_n\} \mapsto \Gamma_{n+1}; L_n} \qquad \text{[A-PAR]}$$

To type check the parallel composition of n processes against a typing context Γ_1 we recursively type check process P_1 resulting in a typing context Γ_2 and a set L_1 of channel end-points. Then, before type checking the process P_2, the rule calls the quotient operation to check that the linear variables used in process P_1 do not remains in the context. The quotient operation ensures that process P_2 does not refers to the linear variables used in process P_1. In other words, the linear channel ends used in P_1 are either delegated (sent on a message) or else turned into an unrestricted type. The result of type checking process P_n is the result of type checking the parallel composition.

When we type check a parallel composition of zero processes the rule return the incoming context and the empty set of variables.

$$\frac{\Gamma_1 \vdash e \mapsto \mathsf{boolean}; \Gamma_2 \quad \Gamma_2 \vdash P \mapsto \Gamma_3; L_1 \quad \Gamma_2 \vdash Q \mapsto \Gamma_4; L_2 \quad \Gamma_3 \equiv \Gamma_4 \quad L_1 \equiv L_2}{\Gamma_1 \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \mapsto \Gamma_3; L_1}$$

[A-IF]

To type check a conditional process if e then P else Q against a typing context Γ_1 the rule verifies that expression e has type boolean, thus obtaining the typing context Γ_2. Γ_2 is used to recursively verify both processes P and Q because only one branch is executed. The output contexts and variable sets must be equal.

$$\frac{\Gamma_1 \vdash A \mapsto \Gamma_2 \qquad \Gamma_2 \vdash P \mapsto \Gamma_3; L}{\Gamma_1 \vdash \mathsf{assert}\ A.P \mapsto \Gamma_3;\ L}$$

[A-ASSERT]

To type check a process of the form assert $A.P$ against a typing context Γ_1 we first subtract the formula A from context Γ_1 to obtain Γ_2. Then we recursively type check process P against Γ_2. The result of this call (a pair composed of a typing context Γ_3 and a set of channel end points L) is the result of type checking the assert process.

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{assume}\ A \mapsto (\Gamma, \mathrm{normalise}\ \ A); \emptyset}$$

[A-ASSUME]

To type check the process of the form assume A we verify whether formula A is well-formed under the incoming context Γ, the rule then adds to the context the formula in normalised form. Given that there is no continuation process the rule returns an empty set of linear variables.

$$\frac{\Gamma_1 \vdash T \qquad \Gamma_1, x\colon T,\ y\colon \mathsf{dualof}\ T \vdash P \mapsto \Gamma_2; L}{\Gamma_1 \vdash \mathsf{new}\ x\ y : T\ P \mapsto \Gamma_2 \div \{x, y\}; L \backslash \{x, y\}}$$

[A-RES]

To type check a channel creation of the form new $xy\colon T\ P$ against a typing context Γ_1 the rule checks whether type T is well-formed and then adds two new entries to the context Γ_1, namely $x\colon T$ and $y\colon$ dualof T, due to the dual behaviour of the channel end-points. The resulting context is used to recursively type check the process P obtaining a new typing context Γ_2 and a set of linear variables in subject position L. Finally, the rule applies the quotient operation $\Gamma_2 \div \{x, y\}$ to ensure that these two end-points are either delegated or turned into unrestricted types in P. It also removes them from L.

$$\frac{\begin{array}{c}\Gamma_1 \vdash x \mapsto q\&\{l\colon T; m\colon U\}; \Gamma_2 \qquad \Gamma_2 + x\colon T \vdash P \mapsto \Gamma_3; L_1 \\ \Gamma_2 + x\colon U \vdash Q \mapsto \Gamma_4; L_2 \qquad \Gamma_3 \equiv \Gamma_4 \qquad L_1 \backslash \{x\} = L_2 \backslash \{x\}\end{array}}{\Gamma_1 \vdash \mathsf{case}\ x\ \mathsf{of}\ l \to P\ m \to Q \mapsto \Gamma_3; L_1 \backslash \{x\} \cup (\mathsf{if}\ q = \mathsf{lin}\ \mathsf{then}\ \{x\} \mathrm{else} \emptyset)}$$

[A-BRANCH]

We address the particular case of branching processes with two branches. The general case should be easy to derive. To type check a branching process of the form $\mathsf{case}\ x\ \mathsf{of}\ l \to P\ m \to Q$ using a context Γ_1, the [A-BRANCH] rule first searches for the type of subject x which must be of the form $q\&\{l\colon T; m\colon U\}$. For each option, l or m, the rule updates the context with $x\colon T$ or $x\colon U$ respectively, and recursively type checks P or Q. Each call results in a pair composed of a typing context and a set of linear variables in subject position L. As in [A-IF] and given that only one process is executed, the rule type checks all alternatives using the same typing context Γ_2 and checks that all calls result in same the context (that is, $\Gamma_3 \equiv \Gamma_4$). Furthermore, all branches must use the same set of linear end-points except for x, that is $L_1 \backslash \{x\} \equiv L_2 \backslash \{x\}$.

$$\frac{\Gamma_2 \vdash x\colon q{+}\{l_i\colon T_i\}_{i\in I}; \Gamma_2 \qquad \Gamma_2 + x\colon T_j \vdash P : \Gamma_3; L \qquad j \in I}{\Gamma_1 \vdash x\ \mathsf{select}\ l_j.P : \Gamma_3; L \cup (\text{if } q = \mathsf{lin} \text{ then } \{x\} \text{else} \emptyset)} \quad \text{[A-SEL]}$$

In order to type check a selection process, $x\ \mathsf{select}\ l_j.P$, given a typing context Γ_1, the rule [A-SEL] searches the type of end-point x and must obtain a type of the form $q{+}\{l_i\colon T_i\}_{i\in I}$ and an arbitrary context Γ_2. The rule then updates context Γ_2 with entry $x\colon T_j$. The obtained context is used to recursively type check the continuation process P, obtaining Γ_3 and L. The result of type checking the selection process is the pair Γ_3 and set L extended with variable x, if q is linear.

To type check an input process prefixed at x, we look for the type of x in Γ. If the type is of the form of $\mathsf{lin}?z : T.U$ then the rule [A-LININP] is used; if the type is of the form of $\mathsf{un}?z : T.U$ then the rule [A-UNINP] is used; if the type is of another form, then the type checking fails.

$$\frac{\begin{array}{cc}\Gamma_1 \vdash x \mapsto \mathsf{un}?z : T.U; \Gamma_2 & \Gamma_2 \vdash x\colon \mathsf{un}?z\colon T.U \equiv U \mapsto \Gamma_3 \\ \Gamma_3, \text{normalise } (y\colon T) \vdash P \mapsto \Gamma_4; L & \text{if } q = * \text{ then } \Gamma_3 \equiv \Gamma_4 \backslash \{y\}\end{array}}{\Gamma_1 \vdash qx?y.P \mapsto \Gamma_4 \div \{y\}; L \backslash \{y\}} \quad \text{[A-UNINP]}$$

To type check an input process prefixed by a variable of an unrestricted type against a typing context Γ_1 the [A-UNINP] rule searches the type of end-point x and must obtain the type of the form $\mathsf{un}?z : T.U$ and context Γ_2. The [A-UNVAR] rule allows to conclude that $\Gamma_1 \equiv \Gamma_2$. Context Γ_2 is used to check whether the continuation type U is equivalent to type $\mathsf{un}?z : T.U$ resulting in a new typing context Γ_3. The rule adds to context Γ_3 the entry $y\colon T$ in its normalised form (thus extracting the formulae from type T). Such context is then used to type check the process P. This results in a new typing context Γ_4 and a set of linear variables in subject position L, both used in the result of the type check of the input process. To ensure that the replicated input process does not use free linear variables the rule compares the contexts Γ_3 and $\Gamma_4 \backslash \{y\}$. Due to rule [A-LINVAR], when a linear variable is used it is removed from the typing context meaning that Γ_4 becomes

different from Γ_3. The end-point x is qualified as unrestricted so it is not necessary add x to set L.

$$\frac{\Gamma_1 \vdash x \mapsto \mathsf{lin}?z : T.U; \Gamma_2 \qquad \Gamma_2, \text{normalise}\,(y\colon T), x\colon [y/z]U \vdash P \mapsto \Gamma_3; L \qquad \text{if } q = * \text{ then } \Gamma_2 \equiv \Gamma_3 \backslash \{y\}}{\Gamma_1 \vdash xq?y.P \mapsto \Gamma_3 \div \{y\}; L\backslash\{y\} \cup x}$$

[A-LININP]

Type checking an input process prefixed by x of linear type $\mathsf{lin}?z : T.U$ uses a rule similar to the previous one. The [A-LININP] rule searches the type of x in Γ_1 obtaining a typing context Γ_2 that does not contain the entry $x\colon \mathsf{lin}?z : T.U$ (due to rule [A-LINVAR]). The rule adds to typing context Γ_2 an entry for the input parameter $y\colon T$, normalised, and another entry $x\colon U$ for the continuation type of x, replacing all occurrences of the bound variable z by y. This context is used to type check the continuation process P resulting in a new typing context Γ_3 and a set of variables in subject position L. Since x is of a linear type we add it to the set L in order to ensure that the linear part of x consumed in P.

$$\frac{\Gamma_1 \vdash x \mapsto \mathsf{un}!z :\ T_1.U; \Gamma_2 \qquad \Gamma_2 \vdash v \mapsto T_2; \Gamma_3 \qquad \Gamma_3 \vdash v\colon T_1 \equiv T_2 \mapsto \Gamma_4 \qquad \Gamma_4 \vdash x\colon \mathsf{un}!z\colon T_1.U \equiv U \mapsto \Gamma_5 \qquad \Gamma_5 \vdash P \mapsto \Gamma_6; L}{\Gamma_1 \vdash x!v.P \mapsto \Gamma_6; L}$$

[A-UNOUT]

Type checking an output process of form $x!e.P$ using the [A-UNOUT] rules requires that the lookup of the type of x in Γ_1 results in a type of the form $\mathsf{un}!z\colon T_1.U$. Given that the type is unrestricted the typing context remains the same, as in [A-UNINP]. The rule calls expression typing to obtain the type T_2 of the value v, and a new typing context Γ_3. The rule compares the obtained type T_2 with T_1 to ensure that the value sent respects the type of x. As in [A-UNINP] the rule verifies if the continuation type U is equivalent to the type $\mathsf{un}!z\colon T_1.U$ and uses the resulting context, Γ_5, to recursively type check the process P. The result of this type checking is the result of type checking the output process, a pair composed of Γ_6 and L.

$$\frac{\Gamma_1 \vdash x \mapsto \mathsf{lin}!z\colon\ T_1.U; \Gamma_1 \qquad \Gamma_1 \vdash v \mapsto T_2; \Gamma_2 \qquad \Gamma_2 \vdash v\colon T_1 \equiv T_2 \mapsto \Gamma_3 \qquad \Gamma_3, \text{normalise}\,(x\colon [v/z]U) \vdash P \mapsto \Gamma_5; L}{\Gamma_1 \vdash x!v.P \mapsto \Gamma_5; L \cup \{x\}}$$

[A-LINOUT]

Rule [A-LINOUT], used to type check output processes prefix by linear channel ends is similar to [A-UNOUT]. Given that the channel end x is linear, we must add its continuation type U to the context after replacing all occurrences of z by e. The resulting context is normalised and used to type check process P. The result of this call is a pair composed of Γ_5 and L. Furthermore, the rule adds channel end x to the set L of linear variables in subject position.

3.4 Derived constructs

In Section 3.2, we introduce new constructs to our language derived from those in the core language. These constructs are process definitions, the type declarations and output and input of multiple values (processes and types). This section shows how we obtained the derived constructs from those in the core language. The purpose of these constructs is facilitate programming, reducing the number of lines of code and errors in SePi programs.

There are two ways to obtain derived constructors: by encoding (or translation) and by admissible rules. The first alternative consists in visiting the AST (abstract syntax tree) and translating the derived constructs to appropriated constructs on the core language thus obtaining a new “core” AST. It requires one extra visit, resulting in runtime costs, and extra code to translate the derived constructs and create a new AST. There is a further disadvantage: each SePi program results in a “core AST” that is used in the validation process. So, eventual errors in this AST become associated not the code present in the source file but else to the translation. For instance, an error in a process definition may be seen by the programmer as an error in an replicated input.

The second alternative leaves the AST unchanged but uses new typing and reduction rules for the new constructs, obtained from the rules in the core language. This alternative dispenses with extra visits and the creation of a new AST. Contrarily to the translation option, errors on SePi source code are associated to the derived constructs, facilitating the production of error messages. However to use this alternative we need to find new type checking and reduction rules (derived from those in the core language) resulting in extra code to type check and interpret the derived constructs. In face to these arguments, we choose the admissible rules alternative to handle the new SePi constructs.

Below we use a translation function, denoted by $[\![\ ]\!]$, that is, $[\![P]\!] = Q$ means that Q is a core process equivalent to P.

Output and input of multiple values. If the expressions are sent or received along unrestricted channels, then the output of multiple values requires the creation of a new channel with a linear type in order to protect the sending operations from unwanted interferences. So, if expressions $e_1, \ldots, e_n$ have types $T_1, \ldots, T_n$, we must create a channel whose end-points have types $\mathsf{lin}?T_1.\ldots.\mathsf{lin}?T_n.\mathsf{end}$ and $\mathsf{lin}!T_1.\ldots.\mathsf{lin}!T_n.\mathsf{end}$. The first end-point is sent to the input process while the second is used to carry expressions $e_1, \ldots, e_n$. After the output of expressions we recursively call the translation function to the continuation process P.

$$[\![x!(e_1, ..., e_n).P]\!] = \mathsf{new}\ y_1\ y_2 \colon \mathsf{lin}?T_1.\cdots.\mathsf{lin}?T_n.\mathsf{end} \qquad x!y_1.y_2!e_1.\cdots.y_2.e_n.[\![P]\!]$$

On the input side, channel x receives, not the values, but a new end-point ready to receive them.

$$[\![x?(y_1, ..., y_n).Q]\!] = x?z.z?y_1. \cdots .z?y_n.[\![Q]\!]$$

As expected the translation of the unrestricted types to send and receive multiple values are as follows:

$$[\![\text{rec } u.\text{un}!(T_1, \cdots, T_n).u]\!] = \text{rec } u.\text{un}!\ (\text{lin}?T_1. \cdots .\text{lin}?T_n.\text{end}).u$$
$$[\![\text{rec } u.\text{un}?(T_1, \cdots, T_n).u]\!] = \text{rec } u.\text{un}?\ (\text{lin}?T_1. \cdots .\text{lin}?T_n.\text{end}).u$$

A simple optimization can be applied when the expressions are sent or received through linear channels. In this case is not necessary to create a new channel. Instead we just replace $x!(e_1, \cdots, e_n)$ by $x!e_1. \cdots .x!e_n$ and adjust the type of x accordingly.

Finally, for the output and input of zero values, $x!()$ and $y?()$, end point x must have type $*!()$ or $\text{lin}!().\text{end}$, and y the dual type. In this case, we may consider an equivalent process that sends a primitive value, $x!5$ for instance, and receiving on a fresh variable $y?z$. Types are adjusted accordingly.

Type declaration. As mentioned in Section 3.2.2 a type declaration type $a = T$ where a may occur in T introduces a type variable a representing the solution of equation $a = T$.

In the case of multiple type declarations we solve the system of equations

$$\begin{cases} a_1 = T_1 \\ \quad \vdots \\ a_n = T_n \quad n > 0, i \neq j \Rightarrow a_i \neq a_j \end{cases}$$

Due to the requirement on contractivity (Section 3.1) and the presence of the recursive type constructor rec, all such systems have a solution. The details are outside the scope of this thesis. For instance the type declarations

```
type T = lin?integer.lin!boolean.U
type U = lin!string.T
```

form a system of equations whose solution is:

$$\begin{cases} T = \text{rec } a.\text{lin?integer.lin!boolean.lin!string}.a \\ U = \text{rec } a.\text{lin!string.lin?integer.lin!boolean}.a \end{cases}$$

Process definition The process definition construct is an abbreviation for a channel creation followed by a replicated input in parallel with the rest of the program.

$$[\![\text{def } X(\vec{p}\colon \vec{T}) = P \;\; Q]\!] = \text{new } XX'\colon *!\vec{T} \;\; *X'?\vec{p}.[\![P]\!] \mid [\![Q]\!]$$

Note that $\vec{p}$ and $\vec{T}$ represent a sequence of zero or more variables and types, respectively, and that the input of multiple values, $X'?\vec{p}$, also requires the translation to the core language. Variable X' is fresh, meaning that there is no other variable named X' in the same program.

Using the algorithmic rules presented in previous section for channel creation [A-RES], the parallel composition [A-PAR] and replicated input [A-UNINP] and unrestricted variables [A-UNVAR] we can obtain a new algorithmic rule for the process definition from the following derivation.

$$\dfrac{\Gamma_1 \vdash *!\vec{T} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash X' \mapsto *?\vec{T};\Gamma_2} \qquad (1) \qquad (2)}{\Gamma_2 \ \vdash \ *X'?\vec{x}.[\![P]\!] \ \mapsto \Gamma_3 \div \{\vec{x}\}; L_1 \backslash \{\vec{x}\}} \qquad \Gamma_3 \div \{\vec{x}\} \div (L_1 \backslash \{\vec{x}\}) \vdash [\![Q]\!] \mapsto \Gamma_4; L_2}{\Gamma_2 \triangleq \Gamma_1, X \colon *!\vec{T}, X' \colon *?\vec{T} \vdash [\![\, * \, X'?\vec{x}.P]\!] \mid [\![Q]\!] \ \mapsto \ \Gamma_4; L_2}}{\Gamma_1 \ \vdash \ \mathsf{def}\ X(\vec{x}\colon \vec{T}) = P\ Q \ \mapsto \ \Gamma_4 \backslash \{X, X'\}; \ L_2 \backslash \{X, X'\}}$$

where (1) and (2) are the undischarged assumptions:

$$\begin{cases} \Gamma_2, \text{normalise}\ (\vec{x}\colon \vec{T}) \vdash [\![P]\!] \mapsto \Gamma_3; L_1 \\ \Gamma_2 = \Gamma_3 \backslash \{\vec{x}\} \end{cases}$$

Noting that $\Gamma_1 \vdash T$ implies $\Gamma_1 \vdash *!T$, and that $\Gamma, X' \colon \mathsf{un}\ p \vdash P$ implies $\Gamma \vdash P$ if X' does not occur free in P (see [36]), we collect the four undischarged assumptions in the above derivation to obtain the following rule.

$$\dfrac{\begin{array}{cc} \Gamma_1 \vdash T & \Gamma_1, X \colon *!\vec{T}, \text{normalise}\ (\vec{x}\colon \vec{T}) \vdash P \mapsto \Gamma_2; L_1 \\ \Gamma_1, X \colon *!\vec{T} = \Gamma_2 \backslash \{\vec{x}\} & \Gamma_2 \div \{\vec{p}\} \div (L_1 \backslash \{\vec{x}\}) \vdash Q \mapsto \Gamma_3; L_2 \end{array}}{\Gamma_1 \ \vdash \ \mathsf{def}\ X(\vec{x}\colon \vec{T}) = P\ Q \ \mapsto \ \Gamma_3 \backslash \{X\}; \ L_2 \backslash \{X\}} \quad \text{[A-DEF]}$$

Multiple declarations. The SePi language allows for multiple *unordered* declarations. In place of channel creation alone in the core language (cf. Section 3.1), SePi allows general processes of the form $D_1 \ldots D_n\ P$ where D_i is either a process definition of the form $\mathsf{def} X(\vec{x}\colon \vec{T}) = P$, a type abbreviation $\mathsf{type}\ x = T$ or a channel creation $\mathsf{new}\ xy\colon T$. All these definitions may be mutually recursive. An example of a sequence of mutually recursive declarations is in next lines.

```
def P x:T = x!true. Q!x
new r w: T
type T = !boolean.U
def Q x:U = x?b. printBoolean!b. P!x
type U = ?boolean.T
P!r | Q!w
```

The translation of these lines results in:

```
new r w: rec a. lin!boolean.lin?boolean.a
new P P1: rec b.un!(rec a.lin!boolean.lin?boolean.a).b
```

```
new Q Q1: rec b.un!(rec a.lin?boolean.lin!boolean.a).b
*P1?z.  z!true.Q!z |
*Q1?z.  z?b.printBoolean!b.P!z |
P!r | Q!w
```

3.5 Programming in SePi

This section presents a few examples attesting the flexibility of the SePi language.

3.5.1 A print server that makes sure values are printed in order

In Section 3.2 we introduced three primitive channels on which string, integer and boolean values may be printed, namely printString, printInteger and printBoolean. However if we want print two or more values in a row without interleaving we need something more that these three channel ends.

For instance given the following definition

```
def printTwoStringsNaive (first: string, second: string) =
  printString!first. printString!second
```

and a client that tries to print three groups of two strings each, as in lines below

```
printTwoStringsNaive!("hello ", "world! ") |
printTwoStringsNaive!("bom ", "dia! ") |
printTwoStringsNaive!("hi ", "there! ")
```

one of the possible outputs is `hello bom hi world! dia! there!`.

In order to overcome this situation we start to define type PrintChannel representing a print channel as seen from the side of the client.

```
1 type PrintChannel = +{printBoolean: !boolean. PrintChannel,
2                        printInteger: !integer. PrintChannel,
3                        printString:  !string.  PrintChannel,
4                        quit: end}
```

The next lines depict an example of a client composed of three concurrent processes, printing two strings each one. The definition printTwoStrings prints two strings in a row.

```
def printTwoStrings (first: string, second: string) =
  printServer?p.
  p select printString. p!first.
  p select printString. p!second.
  p select quit

printTwoStrings!("hello ", "world! ") |
printTwoStrings!("bom ", "dia! ") |
printTwoStrings!("hi ", "there! ")
```

The process definition printTwoStrings asks the server for a print channel and using this channel sends to the server the two strings that he wants to print and selects quit. Before sending to the server the value that he wants to print, the client selects the type

of the value. Each process that calls the printTwoStrings process receives a different print channel p. A possible result of interpreting this code is `hello world! bom dia! hi there!`.

The next lines of code show a print server that prints an arbitrary sequence of values without interleaving from other print commands.

```
5  def printServer printServerImp: *!PrintChannel =
6    def printLoop p: dualof PrintChannel =
7      case p of
8        printBoolean  → p?x. printBoolean!x. printLoop!p
9        printInteger  → p?x. printInteger!x. printLoop!p
10       printString   → p?x. printString!x.  printLoop!p
11       quit          → printServer!printServerImp // recur once done
12
13   new write read: PrintChannel
14   printServerImp!write.
15   printLoop!read
16
17 new printServerImp printServer: *!PrintChannel
18 printServer!printServerImp |
```

The process definition printLoop is responsible for conducting a print session. The server (the printLoop in particular) offers to the client a menu with options to print primitive values (**boolean**, **integer** and **string**) and to close the channel (the option quit). When the client selects quit, the PrintServer is called in order to serve another client. Otherwise, if the client select a printing option, the server receives the value, prints this value and calls the printLoop definition in order to allow to print another value in the same row. On lines 13–15 we create a new print channel. The server keeps the end point read to itself in order to be used in the process definition printLoop while the other is published to be used by a client. On lines 17 and 18 we create a shared print server channel allowing the communication among server and clients.

3.5.2 Channel forwarding

The next example deals with interactions among sender, forwarder and receiver processes, as introduced by Bonelli et al. [6]. The sender process sends to the forwarder process a channel end. In turn, the forwarder process forwards the channel end to the receiver. But what guarantees that a channel-forwarder process does forward the received channel? The answer lies on dependent session types.

The program starts with two type declarations to define the type of the values to be forwarded and the type of "certified" channels to be forwarded. The following declarations create channels to be forwarded (lines 3 and 4), the channel that allow the sender process communicates with the forwarder process (line 5), with end-points toForwarder and forwarderIn, and a channel to ensure the communication between the forwarder and the receiver processes (line 6), with end-points fromForwarder and forwarderOut.

```
1  type ChannelType = end
2  type T = {x: ChannelType | from(x)}
3  new aChannelEnd anotherChannelEnd: ChannelType
4  new anotherChannel v: ChannelType
5  new toForwarder forwarderIn: *!T
6  new fromForwarder forwarderOut : *?T
7  // senders
8  assume from(aChannelEnd) | toForwarder!aChannelEnd |
9  assume from(anotherChannelEnd) | toForwarder!anotherChannelEnd |
10 // receiver
11 fromForwarder*?x.assert from(x).printString!"got it!"
12 // A well behaved forwarder
13 forwarderIn*?x.forwarderOut!x
```

The example is not typable in the type system of Bonelli et al. [6] since channel names may not appear in assertion labels, that is to say, types may only depend on shared names (which are assigned "plain types", ie., **un** types in our terminology).

Before outputting aChannelEnd, the sender process must assume the formula from(aChannelEnd). So, in the presence of a well behaved forwarder, the receiver process asserts the formula from(x), consuming the formula introduced in the type system by the sender process. We now discuss the (incorrect) alternatives. The following line of code shows a forwarder process trying to cheat by sending a different value from x.

```
forwarderIn*?x.forwarderOut!v
```

This process results in two errors:

- the sent value has type ChannelType and the type of channel forwarderOut requires an argument of type {x: ChannelType | from(x)}. This happens because there is no process that assumes the formula from(v).

- when the forwarder receives the channel end, forwarderIn*?x it is introduced the formula from(x) in the type system and there is no process to consume this formula. For each asserted formula, there must be an assume.

Next process depicts another incorrect forwarder process. The forwarder tries to assume a fake value meaning that the original assumption is not matched. Some process must consume the original assumption from(x).

```
forwarderIn*?x.{
    assume from(anotherChannelEnd) |
    forwarderOut!anotherChannelEnd
}
```

There is however one case where the forwarder may send an incorrect value: by asserting formula from(x), which consumes the client assumption, while assuming a new from formula.

```
forwarderIn*?x. assert from(x). {
    assume from(anotherChannelEnd) |
    forwarderOut!anotherChannelEnd
}
```

3.5.3 Request on a channel; respond on a distinct channel

Now we show an example of a request/response between client and server. The example is presented by Gordon and Fournet [17]. This example can be divided in two process definitions, the service and the client. The first reads request messages from a channel and replies on a distinct channel. The second invokes the service and receives the response.

Session types were introduced exactly to allow “request/respond” on the same channel. However we use separate channels to allow a direct comparison with [17]. This is an exercise on refinement types.

The client and the service interact on a linear channel. The reply-to channel is linear, meaning that the server must respond exactly once. The type Request describes a channel that receives a boolean value in form of a request and a channel end to respond with an integer.

```
1  // The type of a request-response as seen by the service
2  type Request =
3      // The request value
4      ?x: {y: boolean | request(y)}.
5      // The "reply-to" channel
6      ?(!{y: integer | response(x, y)}.end).
7      end
```

Each channel end with type Request receives a boolean value, named y, holding request (y) and a channel end-point ready to send an integer such that response(x, y), where x represents the boolean previously received and y the integer value.

```
8  def client(s: dualof Request, query: boolean) =
9      // Create a "reply-to" channel
10     new r1 r2: !{y: integer | response(query, y)}.end
11     // Request and wait for the response
12     assume request(query) |
13     s!query. s!r1. r2?z. {
14         assert response(query, z) |
15         printInteger!z
16     }
```

The client receives a channel with the dual type of Request. It must create a new reply channel, send one end-point to the server and keep the other in order to receive the response. Before requesting (line 13) the client must assume the formula request(query), announcing its intent to request a service. After receiving the response, it must assert response(query, z), making sure that the answer z was received as response to the query.

```
17 def service(s: Request, answer: integer) =
18     // Wait for the request
19     s?query. assert request(query).
20     // Wait for the reply-to channel
21     s?r. {  // Respond
22         assume response(query, answer) |
23         r!answer
24     }
```

In turn, the service process, after receiving the query, asserts the formula request(query), thus making sure that it received a legitimate request. Then receives the "reply-to" channel end. To respond, it firstly assumes response(query, answer) telling that answer is the reply to the request query, and then, it sends the answer to the client.

```
25 new s1 s2: Request
26 service!(s1, 100) |
27 client!(s2, false)
```

The main code of this example only creates a new channel, whose end-points have types Request and **dualof** Request. The first is sent to the server and the second to the client. Because Gordon and Fournet [17] work with classical logic, one can write a typable server that responds twice. In contrast, SePi relies on linear refinements, rendering such sort of servers untypable.

Chapter 4

Implementation

In order to implement the SePi language we use Xtext, a framework that allows to develop new languages while building Eclipse plugins. This chapter starts with the explanation of relevant Xtext features, in Section 4.1. Section 4.2 describes how we implemented the validation phase, by introducing the symbol table and the hierarchies of values, formulae and types. Section 4.3 explains the implementation of the interpreter. Then, Section 4.4 presents some metrics on the implementation (such as lines of code). Section 4.5 briefly describes how we tested the compiler and the interpreter. Finally, Section 4.6 shows how to install and run SePi.

4.1 Xtext and plugin implementation

In order to implement the SePi language and its Eclipse plugin we use Xtext, a language development framework [39]. This framework provides useful mechanisms to write the grammar and to implement the validation and interpretation rules.

We may split the implementation in four parts: parsing, scoping, validation and interpretation. Xtext provides for some important components such as lexer and parser generators, classes to represent the nodes of an abstract syntax tree (AST) and other useful classes for scoping, validation and interpretation, such as appropriated classes to visit the AST. After writing the grammar we generate the Xtext artefacts, obtaining all these classes.

Xtext uses ANTLR (Another Tool for Language Recognition) which implements a LL(*) parser [1]. In order to understand how Xtext works see figure 4.1. The lexer

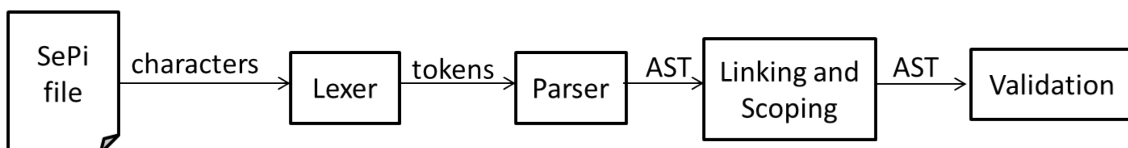
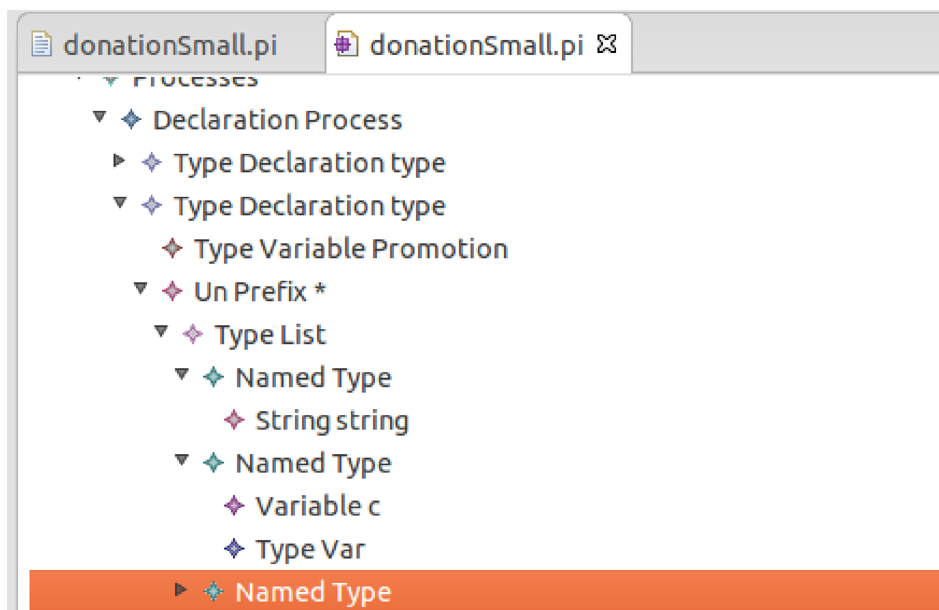


Figure 4.1: Front end of the compiler

receives a sequence of characters (from a SePi file) and converts them in a sequence of tokens. These tokens consist of one or more characters that match a particular lexer rule defined in lexer—grammar file. These tokens are then sent to the parser, which produces a AST. Furthermore, the parser is responsible for the creation of EObjects that constitute the semantic model or AST, but only if there are no syntactic errors in the source file.

When Xtext parses the source code, the result is an AST that can be used in the scoping and linking, validation and interpretation phases. The next figure shows a graphical representation, generated by Xtext, of part of the AST of our running example code (promotion type declaration).



In order to visit the AST, we use a visitor generated by the framework called, in our case, SePiSwitch. This Java class contains a method to visit each kind of node in the AST and, given that SePiSwitch is a generic class, we may define the return type of these methods. As an example, suppose that we want to build the textual representation of an AST. All we have to do is to create a new Java class (subclass of SePiSwitch) and write methods for all sorts of nodes in the AST. That for a type declaration is exemplified below.

```
@Override
public String caseTypeDeclaration(TypeDeclaration td) {
    return "type " +
        caseTypeVariable(td.getTypeName()) +
        " = " +
        caseType(td.getType());
}
```

The first step in checking a syntactically correct program amounts to verify whether all variables are declared. Xtext provides a mechanism to achieve this. First, we identify all the cross-references in the grammar and second, we must find a scope for each reference. The next lines show how cross references are identified in ANTLR using square brackets.

```
Variable:
    name=ID;
Process:
    {Output} channel=[Variable] "!" exprs=ExpressionList
                ("." proc=Process)?
...
```

To find the scope for each reference, we use SePiScopeProvider, one of the classes generated by the framework. In this class, we override the method IScope getScope(EObject context, EReference reference), where the parameter context, in the case above, is an output node and the parameter reference represents the variable whose binder we are looking for. This method returns an IScope that represents the inner most scope of the context. In order to find a scope, we traverse the AST towards its root, looking for variable declarations. Since our language allows nested scopes we must find each scope's outer scope.

If the programmer uses an undeclared variable Xtext issues an error message. However, we must also provide for three implicitly declared variables: printInteger, printBoolean and printString. We create the printing variables, add them to a temporary file and Xtext will find the variable in the AST of this temporary file.

The figure below is an example of an undeclared variable error: in line 36 we typed Read rather than read.

```
34      new write read: PrintChannel    // create a print channel
35      printServerImp!write.    // publish the write end
36      printLoop!Read  // keep the read end
37
```

After Xtext successfully terminates the variable declaration checking (linking and scoping), the framework starts the validation phase. To perform this phase, we use SePiJavaValidator, one of the generated classes by Xtext. This class allows to verify the code in a declarative way. All we have to do is to write the following method and then Xtext visits the SePi node (the root node of any AST in our language) automatically when validation starts.

```
1 @Check
2 public void checkSePi(sePi sepi) { ... }
```

This method uses the visitors described in Section 4.2 to carry out the type checking algorithm.

4.2 The validation phase

This sections describes the main components of the validation phase implementation, such as the symbol table, the type checking process and the hierarchies of values, types and formulae.

4.2.1 The symbol table

We based our symbol table on the one proposed by Appel [2]. We need to use a class Symbol with two methods: a method fresh() that returns a fresh symbol that does not occur in the program and was not generated before and a method symbol(String n) that returns an unique object.

The interface to our symbol table is as follows:

- Type get(Symbol symbol) – returns the Type associated with the symbol symbol in the current scope, or **null**, if the identifier is undefined.

- **void** put(Symbol symbol, Type type) – puts the type type into the table, bound to the symbol symbol.

- **void** update(Symbol symbol, Type type) – updates the entry of symbol with the new type, but checking if the old type and the new type are equal, in the case of unrestricted types. When the types are unrestricted and they are not equal, this operation results in an error.

- **void** beginScope() – starts a new scope and remembers the current state of the table.

- **void** endScope() – closes the current scope, restoring the table to the previous scope.

- **void** delegate(Symbol symbol) – replaces the type associated to symbol by the **end** type.

  The particular behaviour of delegate can be exemplified with the following code taken from our running example (Section 3.2).

```
32 def setup (p: dualof Donation, title: string, date: integer) =
33     case p of
34         setDate  → p?d. setup!(p, title, d)
35         setTitle → p?t. setup!(p, t, date)
36         commit   → if date < 2013 then denied!p else accepted!p
37 def denied (p: dualof Decision) =
38     p select denied.
39     p!"We can only accept 2013 donations\n"
40 def accepted (p: dualof Decision) =
41     p select accepted.
42     promotion!p
```

When the donation campaign is denied, channel p finishes its interaction; when accepted p is delegated to process donation. The [A-IF] algorithmic rule (Section 3.3) makes sure that the symbol tables and the sets of linear variables of both branches are equal. In the last line of the definitionDenied, p is of type **end**. According to rule [A-IF] p must also be of type **end** in the last line of definition accepted. This is the reason why when we delegate a channel (when we remove it from the symbol table), an entry p: **end** is left in its place.

- Set<Symbol> quotient(Set<Symbol> set) – used to check whether the symbols contained in set are all of an unrestricted type, returning a set of symbols. This set contains all the variables that are not unrestricted and it is used for error message purposes.

- Set<Symbol> equivalent(Table other) – compares this table with another one, returning the set of symbols where the comparison fails. The set is then used for error message purposes.

- Set<Symbol> getDomain() – returns a set of all variables (symbols) that appear as keys in the symbol table.

Prior to the validation phase we add to the symbol table the implicitly declared print variables printString, printBoolean and printInteger with types *!**string**, *!**boolean** and *! **integer**, respectively.

4.2.2 Value hierarchy

Figure 4.2 shows the value hierarchy of the compiler. As described in Chapter 3, predicates may refer to values and program variables, which are represented by objects of subclasses of the Value interface.

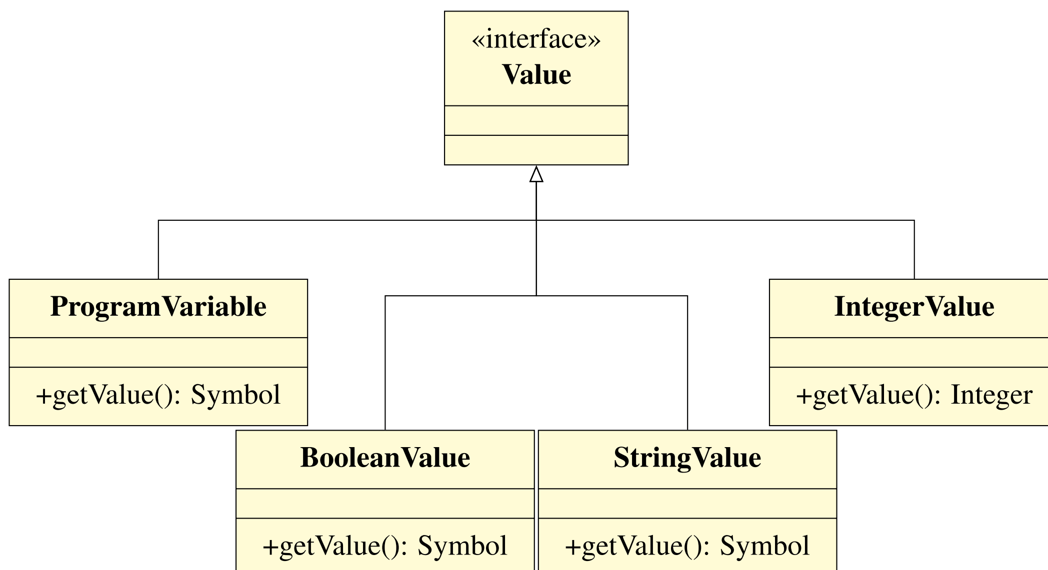


Figure 4.2: The value hierarchy

4.2.3 Formulae hierarchy

Figure 4.3 shows the formulae hierarchy. The abstract class Formula contains the replace

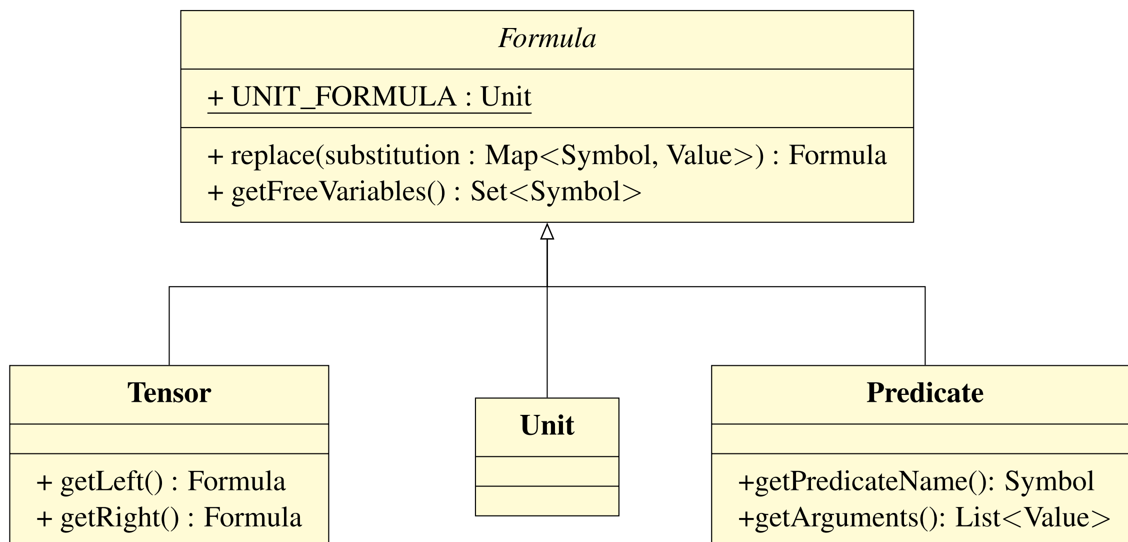


Figure 4.3: The formulae hierarchy

and getFreeVariables methods. Its subclasses are Unit to represent the **unit** formula, Tensor to represent a left * right formula and the Predicate class to represent predicates with form p(v1, ... vn), where the getPredicateName() method returns the symbol p and the getArguments() returns a list with the values v1, ..., vn.

4.2.4 Type hierarchy

Figure 4.4 describes the type hierarchy of our compiler. The classes that represent primitive types Boolean, Integer and String are the simplest classes of this hierarchy. The method hasDual returns **false** and the method dual throws an UnsupportedOperationException. They do not have free variables and the application of a substitution to a primitive type returns the same type. The End class is similar to the primitive types classes, except that its dual() method returns the End type, because **end** is dual of itself. We use the singleton pattern to represent these types where each type is a constant in the class Type.

Class Refinement represents a refinement type of the form {x: T | A}, where the methods getBoundVar(), getType() and getFormula() return the bound variable x, the type T and the formula A respectively. The dual operation is not defined for refinement types; methods hasDual() and dual() behave as for the primitive types.

Class Choice contains a method to check whether the choice is unrestricted or not (**lin** or **un**), a method to check whether the choice is a selection or a branching (+ or &) and a method to obtain all options (a Map from Symbols representing labels to Types). For instance, for the type Decision of our running example, methods isUnrestricted() and isSelect() return **false** and the method options() returns a map with accepted and denied keys.

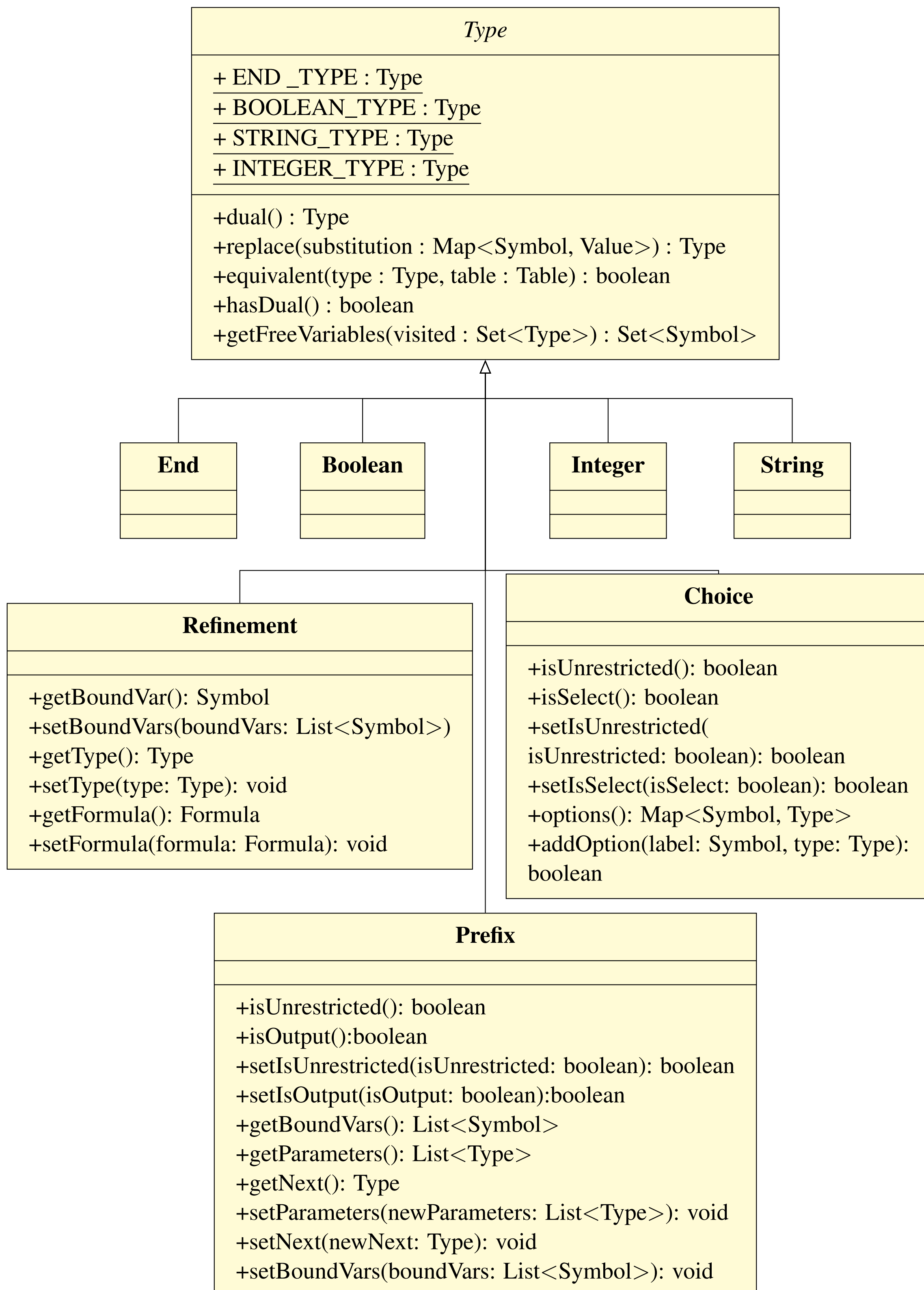


Figure 4.4: The type hierarchy

Class Prefix represents a prefix type where methods isUnrestricted() and isOutput() return true if the type is qualified as linear and if the type represents an output operation, respectively. Methods getBoundVars(), getParameters() and getNext() return the bound variables of the type, its parameters and its continuation type, respectively. Explicit bound variable for type prefixes are optional, as in, e.g., **lin**?**boolean**.**end**. In this case, we generate a fresh variable to represent the bound variable. Each parameter must have a bound variable associated.

The methods of the abstract class Type should be redefined by its subclasses. The replace method returns the type resulting from replacing all symbols by their respective values in the substitution map. This method only modifies formulae.

The equivalent method compares two potentially recursive types. The Table is necessary because when one of the types is a refinement we must call the formulaSubtraction method. The getFreeVariables() method is implemented conform described in Chapter 3. All setter methods in our type classes are necessary due recursion (see Section 4.2.5).

4.2.5 The validation process

This section describes the implementation of the algorithmic rules in Section 3.3. In order to understand the validation phase, consider the following program with mutually recursive declarations (cf. Section 3.4).

```
1 def P x:T = x!true. Q!x
2 new r w: T
3 type T = !boolean.U
4 def Q x:U = x?b. printBoolean!b. P!x
5 type U = ?boolean.T
6 P!r | Q!w
```

The validation phase starts with a few visits to the AST in order to fill the symbol table with all the top level program variables (from process definitions and channel creations) and type variables (from type declarations). Program variables and type variables are stored in different symbol tables.

Mutually recursive declarations are solved in four phases:

Visiting the left hand side of all sorts of declarations. We visit all declarations and for each one we store in the respective symbol table the declared variable. Only, in the case of type declarations we add, to the symbol table, a symbol representing the type variable associated to a skeleton of the type on the right hand side. For instance, if the type is a prefix then, we use the object constructor Prefix(). At the end of this first visit, the map of type variables contains two entries: `(T, Prefix())` and `(U, Prefix())` and the map of program variables contains the entries `(r, null)`, `(w, null)`, `(Q, null)`.

Solving the right hand side of type declarations. We solve the system of equations of

type declarations, as introduced in Section 3.4, using the previous visitor and a new one that visits the right hand side of the type declaration. During this second visit we initialise the types contained in the table. When the second pass is finished, for instance, the mapping of type T is a prefix that represents the type **rec** a.!**boolean**.? **boolean**.a.

Checking the right hand side of channel creations. The visit to the channel creation of this example results in the addition of the entries (r, T) and (w, T') to the symbol table, where T' is the result for the call dual() on type T.

Analysing the right hand side of process definitions. We split this analysis in two visits. The first visits all process definitions and adds to the symbol table the Prefix type corresponding to the definitions. The second visits all process definitions again to add the parameters and respective types to the table and to type check the continuation process.

We also implemented a TypeVisitor, to visit and build new types, a ValueVisitor, to build new values (used in predicates) and a FormulaeVisitor to build new formulae. The types, values and formulae created are those described in the Value hierarchy (4.2.2), Formulae hierarchy (4.2.3) and Type hierarchy (4.2.4) sections.

The typing rules for expressions and processes, described in Section 3.3, are implemented in two new classes—ProcessesTypingRules and ExpressionsTypingRules. When this visitor visits the first output node of the first line of the example presented above, it calls the ExpressionsTypingRules to obtain the type of the argument **true**, the SePi type **boolean**.

The following lines show the simplified implementation of one typing checking rule, in class ProcessesTypingRules—the output rule. We omit some code for simplicity.

```
@Override
public Set<Symbol> caseOutput(Output out) {
    ExpressionsTyping trv = new ExpressionsTyping(this.table,
        super.getValidator());
    Type channelType = trv.caseVariable(out.getChannel());
    if (!isOutputType(channelType, out))
        return new HashSet<Symbol>();
    Prefix subjectType = (Prefix) channelType;
    Symbol subject = Symbol.symbol(out.getChannel().getName());
    EList<Expression> expressions = out.getExpressions();

    if (!sameNumberOfArguments(out, subjectType, expressions))
      return new HashSet<Symbol>();

    // Code to perform substitutions in the continuation type here

    // Code to compare the subjectType with its continuationType if it
    // is unrestricted, or to update the subjectType if it is linear
    // here

    // Recursively type check the continuation process
```

```
    Set<Symbol> result = caseProcess(out.getProc());
    if (!subjectType.isUnrestricted())
        result.add(subject);
    return result;
}
```

4.3 The interpreter

The current SePi interpreter is based on the Turner abstract machine [32], a single processes machine, that allows an implementation on a uni-processor, where concurrent programs are simulated by interleaving the execution of the various processes. This section describes how the interpreter was implemented.

4.3.1 Machine states

A state of the Turner abstract machine is a pair composed of an heap and a run queue. The heap stores channels. Channels are queues of processes waiting to read or to write in channels. The run queue stores processes that are runnable. We use a Closure object to represent a process (an EObject) and its environment (a Map<Symbol, Value> that maps variables to values). We have seen that an input process x?y.P replaces the bound variable y with the received value before continuing with process P. This replacing, in our interpreter, means to add a new entry to the environment of process P, comprising y and its value. For instance, in the below program

```
1 new reader writer: ?string.end
2 writer!"Hello world!" |
3 reader?message.printString!message
```

input reader?message, receives a string from the output process writer!"Hello world!", which means that during the interpretation we create a new Closure with process printString! message (the continuation process) and with an environment containing the entry message : "Hello world!". In line 3 the interpreter uses the string "Hello world!" in place of variable message. The run queue is a Deque<Closure> which stores the closures in the order in which processes should be executed. The heap is a Map<Symbol, Deque<Closure>>. A Deque<Closure> is referred to as a *channel queue*.

4.3.2 The interpretation process

We implemented three classes in our implementation: the Closure, the ExpressionEvaluator and the Interpreter. The ExpressionEvaluator visits and evaluates all expressions sub-trees to obtain integers, strings, boolean values or channels. For instance in the following conditional process,

```
1 new writer reader: !integer.end
2 writer!50.
```

```
3 reader?x.
4 if x < 0
5 then printString!"negative number"
6 else printString!"positive number"
```

the visitor visits the nodes of the boolean expression x < 0 and obtains the boolean value **false** because the evaluation of x results in 50 (defined by the writer process).

The Interpreter class implements the reduction rules of the abstract machine. This class only visits processes nodes in the AST, other kinds of nodes are ignored. At the beginning of the interpretation phase, the interpreter takes the closure at the head of the run queue and executes one reduction step in the process of that closure, and it repeats this process until the run queue is empty. When the visited process is a channel creation, the interpreter creates a new fresh variable to represent the channel, and adds a new entry to the heap with the new variable and the continuation process. The interpreter does no distinguish the two ends of a channel — the two end points are collapsed into a newly created *channel* (that is a program variable). The visit of the assumption process produces no effect and the visit of the assertion of the form **assert** A.P puts a new closure with process P at the head of the run queue. When the process at the head of the run queue is a parallel composition, we add the various parallel processes to the run queue. The visit to a conditional process puts at the head of the run queue the **then** branch if the expression is **true** or the **else** branch otherwise.

The reduction rule of the process definition behaves as the channel creation, but given that the continuation process of the channel creation is a replicated input we put a new closure with the process definition at the heap.

To understand the reduction rules of input and output processes suppose that the head of the run queue contains the closure defined by the environment [x: c; y: c] and the process x?z.P. Using the environment we obtain from x channel c and we search in the heap for an output process ready to send values via channel c, y!v.Q. If there is one, then we create a new closure with the environment [x: c; y: c; z: v] and process P. We add this new closure to the head of the run queue. We also create a new closure for process Q and we add it to the end of the run queue. If, otherwise, the heap does not contain an output process, we remove the closure from the run queue and we add it to the heap. On the other hand, if the closure at the head of the run queue contains an output process y!v.Q and the environment [x: c; y: c; v: 5], and if the heap contains an input process x?z.P, we insert the closure of process Q at the front of the run queue and the closure of the process P to the end, with a new entry z: 5 in the environment. In a given reduction rule, if there is an output closure at the head of the run queue and a replicated input (or a process definition) in the heap, the replicated input remains in the heap (but placed at the end of the channel queue). And conversely, if there is a replicated input at the head of the run queue it remains there.

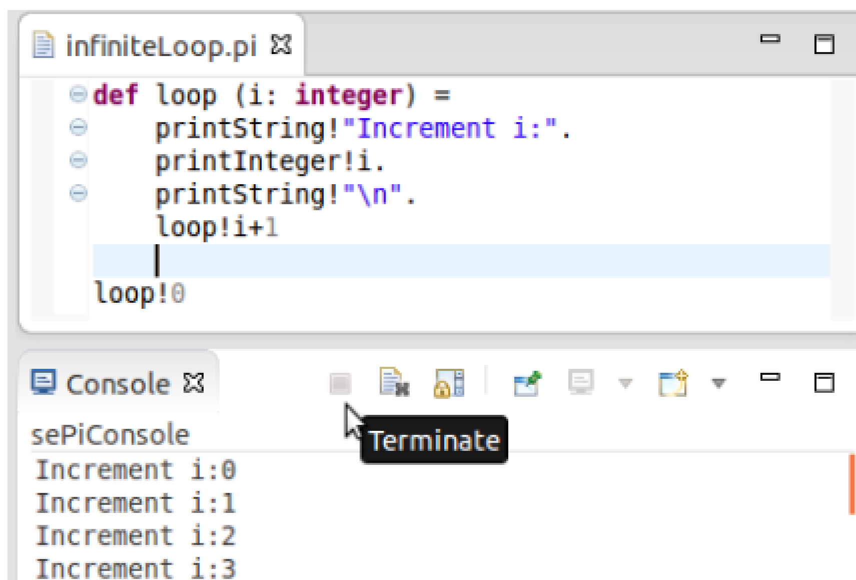


Figure 4.5: Interpreting a program

Turner does not describe reduction rules for selection and branching processes; our implementation follows the ideas of the output/input rules. If the closure at the head of the run queue is a selection process of the form `x` **select** `l.P`, the environment contains the entry `x: c` and the heap contains a branching process waiting to use the channel `c`, then we add new closures with the two continuation processes to the run queue. If the heap does not contain a branching process prefixed by a channel `c` then we put the closure of the selection process in the heap. When the visited node corresponds to a branching process it is necessary to verify if there is a selection process in the heap ready to select a label in the same channel. If yes then the continuation of selection and branching processes are added to the run queue, otherwise, if there is not, we put the closure with the branching process in the heap.

Linear channels are amenable to more efficient implementations. We did not pursued that line of work in this thesis.

Interpreting a program We implement a Java class that calls our interpreter when the programmer requests so. Since our language allows programs that may never terminate, we added to the console a *Terminate* button that stops the execution of the interpreter. Figure 4.5 shows a simple process that represents an infinite loop. This process definition receives an integer and prints it before calling recursively the definition `loop`. To interrupt the execution of this program, the programmer uses the *Terminate* button, as exemplified Figure 4.5.

4.4 Metrics

When we create a Xtext project, the framework generates three new Java projects: the main project that contains the grammar and all the runtime components, such as parser, lexer, linker, validator and interpreter; the tests project and the UI (user interface) project which contains the components for the Eclipse editor and all the other workbench related functionality.

Table 4.1 shows the distribution of the lines of code of these three projects. Note that the grammar was written in a `xtext` file, the SePi code was written in `pi` file and all the other components in Java. The numbers for lines of Java code do not include comments or blank spaces. SePi and Xtext code do so. We may conclude that the largest effort was put in the implementation of the validation phase. All classes for lexer, parser and auxiliary classes of scoping and validation were generated by Xtext.

	Number of files			**Lines of code**		
	source	generated	total	source	generated	total
Main						
grammar	1	0	1	151	0	151
scoping	7	n.a.	n.a.	408	n.a.	n.a.
validation	32	n.a.	n.a.	2.911	n.a.	n.a.
interpretation	5	n.a.	n.a.	566	n.a.	n.a.
other classes	4	n.a.	n.a.	119	n.a.	n.a.
Total	49	482	531	4.155	29.229	33.384
Tests						
Java code	3	2	5	161	41	202
SePi code	287	0	287	9.713	0	9.713
Total	290	2	292	9.874	41	9.915
UI	3	25	28	167	22.678	22.845

Table 4.1: The statistics of the project

4.5 Testing the compiler and the interpreter

We implemented SePi gradually. First we implemented a small language with a few constructs. Then we created a set of SePi programs in order to test this small language. When the language was tested, we introduced new constructs and we added new tests to the first set. In this way we use the new tests to validate the new version of the language and use the old tests for regression testing. We have repeated this process until we have achieved the current version of SePi. Currently we have a set of invalid SePi programs (117 tests), which must result in errors, and a set of valid programs (170 tests), whose result of interpretations must be equal to the expected results (and free from validation

errors). We use JUnit4 to test these files. Xtext provides mechanisms to test the language. We combined these mechanisms with other plugin to write our test suites [38]. In this way we can easily run the test suit and, for each test in the test suit, call some methods that verifies whether the test contains syntactic errors, undeclared variables or validation errors. For valid source code, we can also annotate programs with the expected result, interpret the program and compare the obtained result with the expected.

4.6 Installing & running SePi

This section explains how to program in SePi using the Eclipse plugin or the command line. For further details please visit our website `http://gloss.di.fc.ul.pt/sepi/`.

Developing and running in Eclipse

1. Download an Eclipse with Xtext installed. See how at `http://www.eclipse.org/Xtext/download.html`.
2. Choose **Help** → **Install New Software**. Insert the update site URL (`http://download.gloss.di.fc.ul.pt/sepi/update/`).
3. Restart eclipse when the installation is completed.
4. Create a new project and a new file with the extension `.pi`. At this point the editor asks if you want to add the Xtext nature to your project. Select **yes**.
5. Write your first program. Use the left button → **Interpret** to run your program. See figure 4.6.

Running from the command line This constitutes an alternative to the Eclipse plugin.

1. Download `SePi.jar` available in `http://download.gloss.di.fc.ul.pt/sepi/SePi.jar`.
2. Create a new file with the extension `.pi` and write your program.
3. Open a command line and type `java -jar SePi.jar` *myprogram*`.pi`.

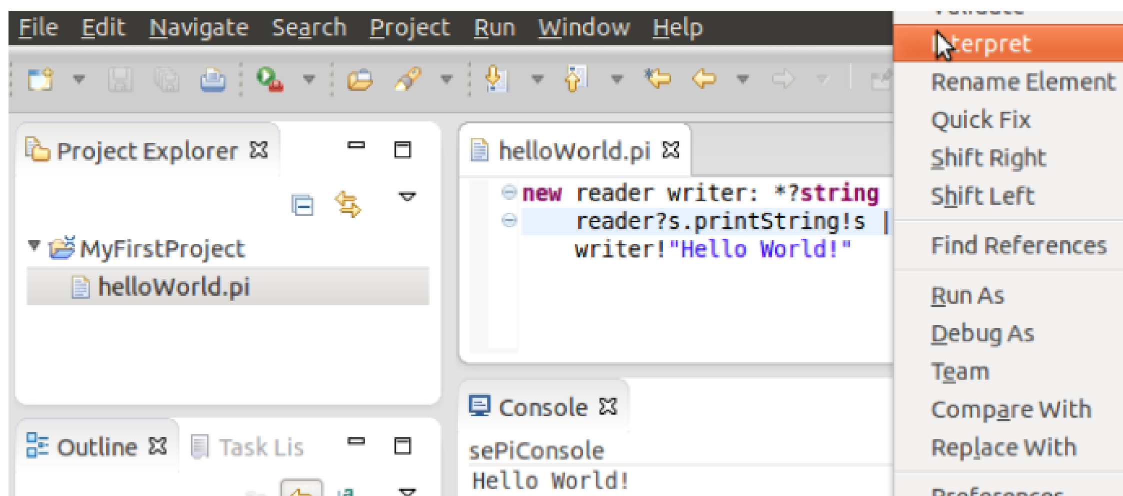


Figure 4.6: A simple hello world program

Chapter 5

Conclusion

We presented a new concurrent, message-passing programming language based on the monadic pi-calculus, called SePi. The language features synchronous, bi-directional channel-based communication between concurrent processes, where the interactions on channels are statically verified against session types. The session types that we use describe the kind and the order of messages exchanged, as well as the number of processes that may share a given channel. In order to obtain a more precise control on the properties of programs, our language includes primitives to assume and assert formulae at the process level and refinement types at the type level. The formal foundation of the language can be found in references [3, 36]. We implemented an interpreter for SePi based on the Turner's abstract machine [32]. Moreover, we wrote an Eclipse plugin that allows to develop SePi code with the usual advantages of an IDE, such as code completion, syntax highlighting, syntactic and semantic validation. According to our knowledge, there is no implementation that combines session types with refinement types, linear or classic.

Through examples, such as the *online donation service*, *print server*, *channel forwarding* and *request and respond on distinct channels*, we show that SePi allows to describe complex interactions between concurrent processes and to specify precise properties of the exchanged values.

There are some aspects of the language that we leave for future work. The current version of SePi only allows predicates over values. Predicates in the form of `p(x + 1)` are not allowed. We plan to add expressions to predicates together with the appropriate theories (e.g. arithmetic). The type system would make use of an SMT solver. SePi can also be extended with a new abbreviation for *session initiation* where a process creates a new channel, sends an end-point and keeps the other to itself (as explained in Chapter 3). Polymorphism and subtyping may be incorporated in future versions of the language. Moreover, we plan to add unrestricted formulae allowing to provide for the persistent availability of resources. We also intend to add to our language an `import` clause allowing the inclusion in code of a different source file thus providing for a limited form of libraries.

Bibliography

[1] Antlr parser generator. http://www.antlr.org/.

[2] Andrew W. Appel. *Modern Compiler implementation in Java*. Cambridge University Press, 2002.

[3] Pedro Baltazar, Dimitris Mostrous, and Vasco T. Vasconcelos. Linearly refined session types. *EPTCS*, 101:38–49, 2012.

[4] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Computer Security Foundations Symposium*, pages 124–140. IEEE, 2009.

[5] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop*, pages 139–152. IEEE, 2006.

[6] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence assertions for process synchronization in concurrent communications. *Journal of Functional Programming*, 2005.

[7] Gérard Boudol. Asynchrony and the pi-calculus. Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

[8] Alexandre Caldeira and Vasco T. Vasconcelos. Bica. http://gloss.di.fc.ul.pt/bica.

[9] Joana Campos and Vasco T. Vasconcelos. Mool. http://gloss.di.fc.ul.pt/mool.

[10] Joana Campos and Vasco T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In *Programming Language Approaches to Concurrency and Communication-cEntric Software*, volume 69 of *EPTCS*, pages 12–28, 2011.

[11] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and Session Types: an Overview. In *Web Services and Formal Methods*, WS-FM'09, pages 1–28. Springer, 2010.

[12] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity OS. *Operating Systems Review*, 40(4):177–190, 2006.

[13] Juliana Franco and Vasco T. Vasconcelos. A concurrent programming language with refined session types. In *Second International Workshop on Behavioural Types*, pages 33–42, 2013.

[14] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, volume 26, pages 268–277. ACM, 1991.

[15] Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Principles of Programming Languages*, pages 299–312. ACM, 2010.

[16] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informaticæ*, 42(2/3):191–225, 2005.

[17] Andrew D. Gordon and Cédric Fournet. Principles and applications of refinement types. Technical Report MSR-TR-2009-147, Microsoft Research, 2009.

[18] Kohei Honda and Gary Brown. Scribble. `http://www.jboss.org/scribble`.

[19] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *Distributed computing and internet technology, LNCS*. Springer, 2011.

[20] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Object-Oriented Programming*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[21] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposym on Programming*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Principles of Programming Languages*, pages 273–284. ACM, 2008.

[23] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *Object-Oriented programming*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[24] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An overview of the singularity project. Technical report, Microsoft Research, 2005.

[25] Robin Milner. *Communicating and Mobile Systems: the pi-calculus*. Cambridge University Press, 1999.

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

[27] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Practical Aspects of Declarative Languages*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[28] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

[29] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, language and interaction: essays in honour of Robin Milner*, pages 455–494. MIT Press, 1997.

[30] Matthew Sackman and Susan Eisenbach. Session types in Haskell: Updating message passing for the 21st century. Technical report, Imperial College, Department of Computing, 2008.

[31] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[32] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[33] Vasco T. Vasconcelos. Typed concurrent objects. In *Object-Oriented Programming*, volume 821 of *LNCS*, pages 100–117. Springer, 1994.

[34] Vasco T. Vasconcelos. TyCO gently. DI/FCUL TR 01–4, Department of Informatics, Faculty of Sciences, University of Lisbon, 2001.

[35] Vasco T. Vasconcelos. Sessions, from types to programming languages. *Bulletin of the European Association for Theoretical Computer Science*, 103:53–73, 2011.

[36] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[37] David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.

[38] Xpect. `http://www.xpect-tests.org/`.

[39] Xtext—language development made easy! `http://www.eclipse.org/Xtext/`.