NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

HUGO FILIPE MINISTRO BAGULHO

BSc in Computer Science and Engineering

A MODULE SYSTEM AND A STANDARD LIBRARY FOR A SESSION-TYPED FUNCTIONAL LANGUAGE

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

April, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

A MODULE SYSTEM AND A STANDARD LIBRARY FOR A SESSION-TYPED FUNCTIONAL LANGUAGE

HUGO FILIPE MINISTRO BAGULHO

BSc in Computer Science and Engineering

Adviser: João Costa Seco
Associate Professor, NOVA University Lisbon

Co-adviser: Bernardo Parente Coutinho Fernandes Toninho
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
April, 2025

A Module System and a Standard Library for a Session-typed Functional Language

Copyright © Hugo Filipe Ministro Bagulho, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.18) [10].

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Bernardo Toninho, for presenting me with the theme for this thesis and for supporting, guiding and encouraging me throughout the development of this work. His patience and dedication during the countless hours spent meeting and debating have been a constant source of inspiration.

On a personal note, I am deeply grateful to my family for staying by my side during the ups and downs of this journey, and encoraging me to maintain the path I've chosen to follow. For all of their support and love, I cannot express how grateful I am.

To my friends, João and Diogo, one of the best things to come out from this journey, I don't know where I would be if I hadn't met them. To André, who has been with me since the 8th grade, who I've grown with and watched grow throughout the years, I couldn't be more proud to be your friend. Thank you for being there for me for all this time and talking some sense into me when I needed it most.

Finally, to my partner, Margarida, who has accompanied me through long days of working on assignments, travels and so much more. Her love and understanding has made me a better person, and for that I am forever in her debt. Thank you for staying by my side through this arduous journey, for supporting me, for believing in me, and for growing with me.

Abstract

The continuous evolution of programming languages is driven by the quest for type safety, expressiveness, and modularity. Among the various paradigms, session types have emerged as a powerful abstraction for structuring communication-based programming, ensuring that interactions adhere to predefined protocols. Previous work has introduced a functional language with session-based concurrency. However, this language lacked a system for defining modules and a standard library to support development.

This thesis aims to address these limitations by implementing a module system in the language, improving its modularity, maintainability, and organization. Additionally, the creation of a standard library provides reusable functionality and assists the user in building programs. In order to demonstrate the capabilities of the module system and of the standard library, a set of examples was developed.

Keywords: Modularity, Functional Language, Session Types, Standard Library

Resumo

A evolução contínua das linguagens de programação é impulsionada pela busca por segurança de tipo, expressividade e modularidade. Entre os vários paradigmas, os tipos de sessão surgiram como uma abstração poderosa para estruturar a programação baseada em comunicação, garantindo que as interações sigam protocolos predefinidos. Trabalhos anteriores introduziram uma linguagem funcional com concurrência baseada em sessão. No entanto, essa linguagem não tinha um sistema para definir módulos e uma biblioteca padrão para dar suporte ao desenvolvimento.

Esta tese visa abordar essas limitações implementando um sistema de módulos na linguagem, melhorando sua modularidade, manutenibilidade e organização. Além disso, a criação de uma biblioteca padrão fornece funcionalidade reutilizável e auxilia o usuário na construção de programas. Para demonstrar as capacidades do sistema de módulos e da biblioteca padrão, um conjunto de exemplos foi desenvolvido.

Palavras-chave: Modularidade, Linguagens functionais, Tipos de sessão, Biblioteca padrão

Contents

List of Figures

Acronyms

ADT Abstract Data Type *(p. 7)*
AST Abstract Syntax Type *(p. 25)*

SCC Strongly Connected Component *(pp. 30, 31)*
SILL Sessions from Intuitionistic Linear Logic *(pp. 1, 19)*

1 Introduction

In the ever-evolving landscape of programming languages, the quest for type safety, expressiveness, and modularity has driven the development of numerous paradigms and tools. Among these, session types have emerged as a powerful abstraction for structuring communication-based programming, ensuring that interactions adhere to predefined protocols.

Previous work [7] presented an implementation of a functional language with session-based concurrency, based on the SILL family of languages [21]. It permits reciprocal dependence between the functional and concurrent layers by dividing them via a contextual monad. A bidirectional typechecking mechanism in this language achieves a delicate balance between type annotation and type inference. This language's code is converted into Go code, which fully utilizes the latter's channel-based concurrency mechanism.

Despite the strengths of session types in enforcing communication protocols, many session-typed languages lack the modularity necessary for large-scale software development. Effective modularity in session-typed languages is crucial for managing complexity, ensuring code reuse, and promoting maintainability. This thesis addresses this gap by demonstrating how a module system was implemented in our language.

In addition to modularity, a well-designed standard library plays a pivotal role in the day-to-day programming experience. A standard library provides a collection of pre-defined modules and functions that simplify common tasks, promote best practices, and ensure consistency across codebases. By providing these tools out-of-the-box, the standard library will reduce the need for developers to reinvent common functionalities, allowing them to focus on the unique aspects of their applications. Thus, this thesis also presents the creation of a standard library that leverages the implemented module system.

1.1 Contributions

The programs in our language were previously comprised of a single file, which made it difficult to manage large projects. Therefore, our motivation in building a module system was to improve the maintainability, modularity and organization of the user's code. For

this purpose, we built a module system and provided a standard library to auxiliate the user in building its programs.

1.2 Document Outline

We start by addressing the need to implement a module system in our language, and presenting several languages similar to ours that already have this feature (Chapter 2). Then, in Chapter 3 we describe the syntax, types and compiler in our language previous to the implementation of the module system.

In Chapter 4 we present the changes and additions made to the language, namely the module system and the standard library. Afterwards, in Chapter 5, we present the results obtained from the implementation of the module system and the standard library.

Finally, in Chapter 6 we present the conclusions of our work and suggest future work that can be done to improve the language.

2

Background and Related Work

2.1 Modularity

Modularity is a programming language design technique used with the purpose of dividing a system into several separate modules or components and each of these modules corresponds to a specific functionality and operates separately from other modules. The use of this technique brings many benefits to the language it is implemented in [11], them being:

- **Simplicity**: the use of modules simplifies the design of the language, as well as its development, testing and maintenance, by having the functionalities in different modules;
- **Organization**: modules allow for having different functionalities placed in different modules, making it easier to organize the language by having different functionalities separated, with unique names that discriminate what each one does. This way, when updating a certain module, the developer only needs to find its location;
- **Parallel Development**: with the use of modules, the developers of the language are able to write and update the code of the modules at the same time without interfering with each other, enhancing collaboration among team members;
- **Security**: with the isolation of functionalities into separate modules, in the case that we find a vulnerability in one of the modules, we limit the exposure of other modules to the vulnerable one, which contains potential risks.

In spite of all of these advantages, using a modular architecture in a programming language can also have some disadvantages, some of them being:

- **Integration Issues**: when creating new modules, the process of integrating them within the system may be difficult and require a lot of work to guarantee that every module is compatible. If one of the modules is not compatible with the others, it

may result in a more expensive implementation by requiring more effort and time to troubleshoot and find the errors;

- **Higher Cost**: the cost of building a programming language with a modular architecture can be quite expensive due to the time and effort put in by the developers of the language and also by the resources that need to be designed and implemented;

- **Performance Overhead**: the modules developed in the programming language may require additional communication between different modules and can increase the memory usage of the system, leading to performance overhead.

When deciding wether or not to implement a modular architecture in our programming language, we pondered the advantages and disadvantages presented above and how their implementation in our programming language will affect it. Despite having some valid points regarding the disadvantages, we believe the advantages outweight them. Previously, our language was comprised of a single file, where a program was a set of several definitions with a special definition named `main`. Having different functionalities placed in modules is beneficial to our language, since it would simplify the structure of the system and allow us to organize our code better. Considering that our language is new and simple, it would benefit a lot from the use of this technique. With this implementation in our language, the increased complexity becomes an improvement for the language, developing and improving it, while the other disadvantages become almost irrelevant. Therefore, there is a clear profit from using the modular architecture.

2.2 Languages with Modules

Modular architecture is used by many other languages in their code. However, the ones we'll further analyze are Haskell and Ocaml because they are functional languages.

2.2.1 Haskell

Haskell is a purely functional programming language which uses type inference and is designed for teaching and research, being one of the firsts to develop programming language features like type classes [9].

2.2.1.1 Modules

In Haskell, the creation of modules is carried out in files with the extension "`.hs`", where we first define the module name and the list of operations it encompasses, and then we define those operations, as seen in the Example 2.3.1.1 below:

```
-- Definition of module MathOperations
module MathOperations
  ( add, -- exporting add function
    subtract -- exporting subtract function
  ) where

-- Definition of function to add two numbers
add :: Num a => a → a → a
add x y = x + y

-- Definition of function to subtract two numbers
subtract :: Num a => a → a → a
subtract x y = x - y
```

Example 2.3.1.1: Creating a module

When importing in Haskell, we can *import* either the entirety of the module or specific functions. To import the created module we use the statement `import` followed by the name of the module created, as presented in the Example 2.3.1.2 below:

```
-- Importing MathOperations module
import MathOperations(add, subtract)

-- Using imported functions
main :: IO ()
main = do
  print (add 5 3)
  print (subtract 5 3)
```

Example 2.3.1.2: Importing the created module

Another way to import a module is by using the form `import qualified module as alias`, where `module` is the name of the module we want to import, and `alias` is the name we want to give to the imported module. Instead of importing the entire module, we can also select which functions we wish to import (Example 2.3.1.3). This operation can be done in two different ways: the first is by using the statement `import`, followed by the name of the module, the keyword "`hiding`" and the names of the functions we wish not to import, in parentheses. The second and the one used in the example bellow, where we use the statement `import`, followed by the name of the module created, succeeded by the names of the functions we want to import in parentheses:

```
-- Import the add function from MathOperations module
import Mathoperations (add)

main :: IO ()
main = print (add 5 3)
```

Example 2.3.1.3: Importing a function from the created module

In Haskell we can *export* functions (Example 2.3.1.4) and types (Example 2.3.1.5), and both are defined in a very similar way. To export functions, we first choose a module name

and then we list the functions in parentheses. The process is the same for exporting types, as we choose a module name, list the types in parentheses after the module name, and define those types.

```
-- Definition of module ExportFunctions
module ExportFunctions
  ( multiply, -- exporting multiply function
    divide -- exporting divide function
  ) where

-- Definition of function to multiply two numbers
multiply :: Num a => a → a → a
multiply x y = x * y

-- Definition of function to divide two numbers
divide :: Fractional a => a → a → a
divide x y = x / y
```

Example 2.3.1.4: Exporting a function

```
-- Definition of module ExportTypes
module ExportTypes
  ( Color(..), -- exporting Color type and constructores
    Shape -- exporting Shape type
  ) where

-- Defining Color type with constructores
data Color = Red | Green | Blue deriving Show

-- Defining Shape type
data Shape = (Int, Int)
```

Example 2.3.1.5: Exporting types

To use the exported functions and types above, we first *import* the modules created for the functions and for the types using the `import` statement and then we are able to use whatever functions and types we want that exist in those modules by simply calling them in our code, as seen in the Example 2.3.1.6:

```
import ExportFunctions (multiply, divide)
import ExportTypes (Color(..), Shape)

main :: IO()
main = do
  print (multiply 3 4)
  print (divide 4 2)
  print Red
  let rectangle: Shape = (3, 4)
  print rectangle
```

Example 2.3.1.6: Using the exported functions and types

The functions used in Haskell modules must utilize *signatures*, which are configurated

in files with the extension ".hsig". These signatures are crucial for specifying the expected types and functions that a module should include, without the need to detail their implementation. In the ".hsig" file, we use the `signature` statement, followed by the name of the file. Then, we declare the expected types of each function without implementing it, as in Example 2.3.1.7:

```
-- File: MathOperations.hsig
signature Mathoperations where

-- Expected function: add
add :: Int → Int → Int

-- Expected function: subtract
subtract :: Int → Int → Int
```

Example 2.3.1.7: Creating the module signatures

Upon creating the module signatures, these must still be used in the development of the module. Before each function implementation, we must present the signature corresponding to its function, as seen in Example 2.3.1.8. These signatures are used during the import process, after importing them to the desired file.

```
-- File: MathOperations.hs
module MathOperations where

-- Implementing add function
add :: Int → Int → Int
add x y = x + y

-- Implementing subtract function
subtract :: Int → Int → Int
subtract x y = x - y
```

Example 2.3.1.8: Implementing the module signatures

Modules in Haskell can also be used to create and build Abstract Data Type (ADTs), which are types whose representation is hidden but contain associated operations. Haskell's module system supports the definition of ADTs. All of the operations on the ADT are made at an abstract level, free from the representation of the type. The example below represents a parameterized data type of a `Tree`. These parameterized data types can be viewed as an abstract type, since they abstract some parts of the data type. It is only when a user uses the ADT that they indicate which type they want to use, therefore different users can use different types within the `Tree` without changing its implementation. The Abstract Data Type `Tree` has the structure in Example 2.3.1.9: and can be used like in Example 2.3.1.10:

```
data Tree a = Nil
            | Node { left :: Tree a,
                     value :: a,
                     right :: Tree a }
```

Example 2.3.1.9: Structure of the Abstract Data Type Tree

```
three_number_tree :: Tree Integer
three_number_tree :: Node (Node Nil 1 Nil) 2 (Node Nil 3 Nil)
```

Example 2.3.1.10: Usage of the Abstract Data Type Tree

Haskell makes use of polymorphism [16] by having values that can have more than one type, making this a key feature in the language. There are two major types of polymorphism in Haskell: *parametric* polymorphism and *ad-hoc* polymorphism. In parametric polymorphism, the type of a value contains one or several type variables which allows the value to take on any type that emerges from replacing those variables with concrete types. In more specific terms, a function with the signature $id :: a \rightarrow a$ has an unconstrained type variable a that can be used in several contexts, where they can require a $Char$ ($Char \rightarrow Char$), an $Integer$ ($Integer \rightarrow Integer$), or a $Bool$ (($Bool \rightarrow Maybe\ Bool$) $\rightarrow$ ($Bool \rightarrow Maybe\ Bool$)), as well as any of the other existing types. The same way, the polymorphic function $map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$ can operate on any function type, since the empty list $[] :: [a]$ belongs to every function type. To prevent any errors, once a single type variable appears multiple times, that same variable must contain the same type every time it is used, and regardless of the type used for the variable, it must always offer the same behavior. The name of the property that guarantees this is *parametricity*. For example, the result type and argument type of id must both have the same type, as well as the input and output types of the function given to map, which must have the same type as the list types.

In ad-hoc polymorphism, a value can adopt any one of many types since it has been given a separate definition for each different type, i.e. there are multiple definitions of the same function for different types. In Haskell, this is made possible because of the *type classes* and class instances. Haskell also allows to define class instances with polymorphic types, which can be parametical or ad-hoc. This means that if a type variable a that is an instance of a type class, then $[a]$ will also have an instance, as well as $[[a]]$, and so on.

2.2.1.2 Type Classes

Haskell's type classes [22] have many similarities with interfaces, where they define a set of values or methods by their type signature. They also provide a default recursive implementation of the methods of the type class based on each of the methods or values presented earlier. To demonstrate this functionality, we will present the `Eq` [5], `Ord` [14], `Read` [18] and `Show` [19] type classes. The `Eq` type class defines equality and inequality and is defined as follows:

```
1 class Eq a where
```

```
2   (==) :: a → a → Bool
3   (/=) :: a → a → Bool
4
5   x /= y = not (x == y)
6   x == y = not (x /= y)
```

The default implementation of the behavior for equality (**line 2**) and inequality (**line 3**) is presented in **lines 5** and **6**, by negating each other. The use of the `Eq` type class is made by using *instances* of this class. To instanciate the `Eq` type class for `Float` values, by defining the equality function for the given type, we immediately obtain the definition of inequality, due to the default implementation of the `Eq` type class. The instantiation of `Eq Float` is presented as follows:

```
1 instance Eq Float where
2   (==) = eqFloat
3
4   eqFloat :: Float → Float → Bool
5   (F# x) `eqFloat` (F# y) = isTrue# (x `eqFloat#` y)
```

In this instance, first we define the behavior of the equality, which will receive two `Floats` and return a `Bool` (**lines 2** and **4**), and then implement that behavior (**line 5**). There can only exist one instance of the `Eq` type class for each of the existing types. The `Ord` type class is used for totally ordered datatypes. It depends on the `Eq` type class (**line 1**) and defines the behavior of the methods `compare` (**line 2**), <, >, <=, >= (**line 3**), `max` and `min` (**line 4**). The default implementation of these methods is presented in the lines that follow, and rely on one another:

```
1  class  (Eq a) => Ord a  where
2    {-# MINIMAL compare | (<=) #-}
3    compare              :: a → a → Ordering
4    (<), (<=), (>), (>=) :: a → a → Bool
5    max, min             :: a → a → a
6
7    compare x y = if x == y then EQ
8                  else if x <= y then LT
9                  else GT
10
11   x <= y = case compare x y of { GT → False; _ → True }
12   x >= y = y <= x
13   x > y = not (x <= y)
14   x < y = not (y <= x)
15   max x y = if x <= y then y else x
16   min x y = if x <= y then x else y
```

To instanciate the `Ord` type class, we must follow the rule of implementing, at the least, the functions `compare` or <=, at the minimum (**line 16**). All of the other functions can be deducted through the implementation of one of these functions:

```
1 instance Ord Float where
2   (F# x) `compare` (F# y)
```

```
3       = if      isTrue# (x `ltFloat#` y) then LT
4         else if isTrue# (x `eqFloat#` y) then EQ
5         else                                  GT
6
7   (F# x) <  (F# y) = isTrue# (x `ltFloat#` y)
8   (F# x) <= (F# y) = isTrue# (x `leFloat#` y)
9   (F# x) >= (F# y) = isTrue# (x `geFloat#` y)
10  (F# x) >  (F# y) = isTrue# (x `gtFloat#` y)
```

The `Read` type class parses `Strings` and produces values. It contains four methods: `readsPrec`, `readList`, `readPrec` and `readListPrec`, whose behavior and default recursive implementations are presented below:

```
1  class Read a where
2    {-# MINIMAL readsPrec | readPrec #-}
3
4    readsPrec     :: Int → ReadS a
5    readList      :: ReadS [a]
6    readPrec      :: ReadPrec a
7    readListPrec  :: ReadPrec [a]
8
9    readsPrec     = readPrec_to_S readPrec
10   readList      = readPrec_to_S (list readPrec) 0
11   readPrec      = readS_to_Prec readsPrec
12   readListPrec  = readS_to_Prec (\_ → readList)
```

The minimal requirement for instanciating the `Read` type class is the implementation of the functions `readsPrec` or `readPrec`. The instance of the `Read` class for `Bool` values implements the functions `readPrec`, `readListPrec` and `readList`, and obtains the remaining function by deduction.

```
1  instance Read Bool where
2    readPrec =
3      parens
4      ( do L.Ident s ← lexP
5        case s of
6          "True"  → return True
7          "False" → return False
8          _       → pfail
9      )
10
11   readListPrec = readListPrecDefault
12   readList     = readListDefault
```

The `Show` type class is used to convert a value to a readable `String`. It contains three methods: `showsPrec`, `show` and `showList`. The method `showList__` is an auxiliary method to help implement the default recursive method `showList`:

```
1  class  Show a  where
2    {-# MINIMAL showsPrec | show #-}
3
```

```
4   showsPrec :: Int → a → ShowS
5   show      :: a   → String
6   showList  :: [a] → ShowS
7
8   showsPrec _ x s = show x ++ s
9   show x          = shows x ""
10  showList ls   s = showList__ shows ls s
11
12  showList__ :: (a → ShowS) → [a] → ShowS
13  showList__ _     []     s = "[]" ++ s
14  showList__ showx (x:xs) s = '[' : showx x (showl xs)
15            where
16              showl []     = ']' : s
17              showl (y:ys) = ',' : showx y (showl ys)
```

In order to instanciate the `Show` type class with a `Char` type we must implement at least one of the two methods: `showsPrec` or `show`. The instance `Show Char` implements the `showsPrec` and `showList` methods and the other methods are derived from this implementation. The functions used to help implement these methods are utility functions that support the type class `Show`:

```
1  instance Show Char where
2    showsPrec _ '\'' = showString "'\\''"
3    showsPrec _ c = showChar '\'' . showLitChar c . showChar '\''
4
5    showList cs = showChar '"' . showLitString cs . showChar '"'
```

2.2.2 Ocaml

Ocaml is a general purpose programming language with a focus on security and expressiveness, along with a complex module system that enables parameterizing a module over several other modules and arranging modules hierarchically.

2.2.2.1 Modules

In this module system [13], modules can become *submodules* of another module, just like directories in a file system. Therefore, every code piece belongs to a module, meaning that these are file-based modules. The file-based modules are defined have the extension "`.ml`", while the interface files have the extension "`.mli`". The module files define functions, values and type constructors, while the file interface defines the signatures of those operations. In Example 2.3.2.1.1 below we present an implementation of the function `hello` in the file `hello.ml`:

```
let helloMessage = "Hello World"
let hello () = print_endline message
```

Example 2.3.2.1.1: File module hello.ml

The next example demonstrates how to use the module functions in another file. This is done by referencing the file module we are using with a capital letter, followed by a "." and the name of the operation we wish to use, as seen below:

```
let () = Hello.hello ()
```

Example 2.3.2.1.2: Usage of file module hello.ml

The interfaces for the file modules contain the signatures of the operations present in those files, and those distinguish between public and private operations. There are two cases for this specification: the first is the case where an interface is not provided and, by default, all of the operations are public; the second is the case where an interface is explicitly provided, allowing to discern between public and private operations. To make certain operations private, we simply need to hide the signatures for those functions in the interface, while presenting the signatures of all the functions we wish to maintain public. Example 2.3.2.1.3 represents the interface for the `hello.ml` file, where the value `helloMessage` is commented, and therefore private.

```
(** val helloMessage : string *)
val hello : unit → unit
```

Example 2.3.2.1.3: File module interface hello.mli

Submodules in Ocaml originate from specifying a module within a file, making it a submodule. To define a submodule, we use the keyword "`struct`" and followed by the name of the submodule, as seen in Example 2.3.2.1.4:

```
module Goodbye = struct
  let goodbyeMessage = "Goodbye World"
  let print () = print_endline goodbyeMessage
end
```

Example 2.3.2.1.4: Submodule Goodbye in file hello.ml

The usage of these submodules is very similar to the usage of file modules. In this case, we must first reference the file module name where the submodule is defined, followed by a "." and the name of the submodule, followed by another "." and the operation we wish to use, as seen in Example 2.3.2.1.5:

```
let () = Hello.Goodbye.print ();
```

Example 2.3.2.1.5: Usage of submodule Goodbye

Submodules may also be presented with their respective signatures, which are made before the implementation of the module, with the use of the keyword "`sig`". By not presenting the signature of the value `goodbyeMessage`, it becomes private and inaccessable by other modules or files, while the function `print` is public. Example 2.3.2.1.6 presented below represents the definition of submodule Goodbye and its respective signature:

```
module Goodbye : sig
  val print : unit → unit
end = struct
  let goodbyeMessage = "Goodbye World"
  let print () = print_endline goodbyeMessage
end
```

Example 2.3.2.1.6: Submodule Goodbye with signature

2.2.2.2 Functors

Ocaml allows for the creation of functions that, instead of receiving and outputing values, receive and output modules and perform operations on those modules. These functions are called `functors` [12]. To explain functors better, we will refer to an already existing module `Set`, with a functor named `Make`. The `Set` module handles sets, as well as union, intersection, and difference on sets. The functor `Set.Make` provided by the `Set` module is necessary to create a set module for a given element type. The interface presented below is a simplified version of the `Set.Make` functor. In this interface, the functor receives a module with signature `OrderedType` and returns a module with signature `Set.S`. The module type `OrderedType` requires a type `t` and a function `compare`, which are used to compare the elements of the set.

```
module type OrderedType = sig
  type t
  val compare : t → t → int
end

module Make : functor (Ord : OrderedType) → Set.S
```

The `Set.Make` functor is used to create a set. The example below represents the usage of the functor `Make`:

```
module StringCompare = struct
  type t = string
  let compare = String.compare
end

module StringSet = Set.Make(StringCompare)
```

Functors can be used to parameterize modules. It is possible because a functor is similar to a module, except it needs to be applied to one, turning it into a module. This module parametrisation is the case for sets, maps and hash tables. If we provide to the functor a module that implements what is expected, described by the parameter interface, it will return the expected result module. The `Set.Make` and `Map.Make` have the same interface, which was presented previously, and `Hashtbl.Make` has the following interface:

```
module type HashedType = sig
  type t
  val equal : t → t → bool
```

```
  val hash : t → int
end
```

The result of the functors `Set.Make`, `Map.Make` and `Hashtbl.Make` is the expected module that corresponds to the interfaces `Set.S`, `Map.S` and `Hashtbl.S`, respectively. Each of these interfaces contains a type and the associated functions. Obtaining parametrised data structures is the reason we write functors. To get these data structures, we can write our own functors. To explain this concept, we will present the example of `heap` data structures. These data structures include binary, leftist, binomial or Fibonacci heaps, and we will focus on binary heaps. The signature of the module that is received as a parameter in the functor is the module type `OrderedType`, the same as in `Set.Make` and `Map.Make`. The signature of the module that is the result of the functor will be a module type `HeapType`. Any heap must have the means necessary to compare the elements they contain, therefore it must provide at least the interface of the `HeapType` below. It must contain two types `elt` and `t`, an `empty` function, an `is_empty`, an `insert` function, a `merge` function, a `find` function, and a `delete` function.

```
module type OrderedType = sig
  type t
  val compare : t → t → int
end

module type HeapType = sig
  type elt
  type t
  val empty : t
  val is_empty : t → bool
  val insert : t → elt → t
  val merge : t → t → t
  val find : t → elt
  val delete : t → t
end
```

The `Binary` module receives the `OrderedType` module type and outputs the `HeapType` module type. It also implements the functions of the `HeapType` signature:

```
module Binary(Elt: OrderedType) : HeapType = struct
  type elt (* Add your own type definition *)
  type t (* Add your own type definition *)
  (* Add private functions here *)
  let empty = failwith "Not yet implemented"
  let is_empty h = failwith "Not yet implemented"
  let insert h e = failwith "Not yet implemented"
  let merge h1 h2 = failwith "Not yet implemented"
  let find h = failwith "Not yet implemented"
  let delete h = failwith "Not yet implemented"
end
```

Modules have the ability to be extended by using a Standard Library functor. In the example below, we present the module `String`, which uses the "`include`" keyword to import the `String` type, and use the submodule `Set` to extend the `String` module. The submodule `Set` is the output result of the functor `Set.Make` that receives as input the imported type.

```
module String = struct
  include String
  module Set = Set.Make(String)
end
```

2.2.3 FreeST

FreeST [1] is an interpreted programming language that offers session typed concurrency, and is largely inspired by Haskell, while taking some inspiration from Ocaml as well. FreeST doesn't have a modular system that we can study and take inspiration in, but it does have a functionality which is the use of IO functions that perform operations through channels instead of using the IO monad, implemented by [2]. This concept will be valuable for the construct of a standard library for our language.

To understand how the IO operations work, we'll first present the session types (channels) for the standard IO and the signatures of the file IO that produce those channels. Since FreeST has different session types for input and output, so it creates two different types for each, which are presented below:

```
type OutStreamProvider : *S = *? OutStream
type OutStream : 1S = +{ PutChar : !Char   ; OutStream
                       , PutStr  : !String ; OutStream
                       , PutStrLn : !String ; OutStream
                       , Close   : End}

type InStreamProvider : *S = *? InStream
type InStream : 1S = +{ GetChar : ?Char   ; InStream
                      , GetLine : ?String ; InStream
                      , IsEOF   : ?Bool   ; InStream
                      , Close   : End }
```

The `OutStreamProvider` and `InStreamProvider` types are shared versions of the standard IO for output and input, respectively. These types use the `OutStream` and `InStream` session types, which are linear versions of the standard IO. These last two types are recursive and are coded by using an internal choice. For example, the `OutStream` session type provides four operations. Three of these operations will output either a `Char`, in the case of the `PutChar` operation, or a `String`, in the case of `PutStr` or `PutStrLn`, and receive an `OutStream` channel, turning these operations recursive. The `Close` operation terminates the recursive process. The standard IO channels can be accessed globally through their functions. File channels, on the other hand, must be opened for each file. Just as in the standard IO's input and output, write file and read file operations have

different signatures. These operations return `OutStream` and `InStream` types, respectively. The signatures presented below refer to standard IO (`stdout, stderr` and `stdin`) and file IO (`Filepath, openWriteFile, openAppendFile` and `openReadFile`) signatures.

```
stdout , stderr : OutStreamProvider
stdin : InStreamProvider

type FilePath = String
openWriteFile , openAppendFile : FilePath → OutStream
openReadFile : FilePath → InStream
```

The following functions are applied over FreeST's channels by using an abstraction of channel interaction and represent operations over the `stdout` and `stdin` channels. These functions always output the resulting channel, with the exception of `hClose`, which closes the channel.

```
hPutChar : Char → OutStream → OutStream
hPutStr , hPutStrLn : String → OutStream → OutStream
hPrint : ∀ a :1T . a → OutStream → OutStream

hGetChar : InStream → ( Char , InStream )
hGetLine , hGetContents : InStream → ( String , InStream )

hCloseIn : InStream → ( )
hCloseOut : OutStream → ( )
hIsEOF : InStream → ( Bool , InStream )
```

The functions presented above only work for the linear session type. So, for the shared session type, we span over `OutStreamProvider` and `InStreamProvider` typed channels. These functions perform a single iteration and then close the channel. The operations over `stdout` and `stdin` channels for the shared session types are as follows:

```
hPutChar_ : Char → OutStreamProvider → ( )
hPutStr_ , hPutStrLn_ : String → OutStreamProvider → ( )
hPrint_ : ∀ a :1T . a → OutStreamProvider → ( )

hGetChar_ : InStreamProvider → ( )
hGetLine_ , hGetContents_ : InStreamProvider → ( )
```

FreeST then implements abstractions to make IO programming easier. The abstract interaction with the `stdout`, `stdin` and `file` channels presents the following signatures:

```
putChar : Char → ( )
putStr , putStrLn : String → ( )
print : ∀ a :1T . a → ( )

getChar : Char
getLine , getContents : String

writeFile , appendFile : FilePath → String → ( )
readFile : FilePath → String
```

To support the interface that the programmer will use, FreeST provides language primitives which are implemented in the interpreter. `Stdout` and `stderr` will have one primitive each, while `stdin` has two primitives, one to read a `Char` and the other to read a `line`. For files, in contrast with standard IO, FreeST presents more than one primitive for some of the functions. To interact with a file we must keep track of its `Handle`. `Handles` are a Haskell functionality to track a file. FreeST copies this functionality by creating a data structure that represents a `Handle` named `FileHandle`. Since there is no function to turn, for example, an `Int` into a `String`, FreeST creates the `show` primitive and the four `read` primitives, one for each type. The signature of the primitives that support the programmer interface for IO are as follows:

```
__putStrOut : String → ( )
__putStrErr : String → ( )

__getChar : ( ) → Char
__getLine : ( ) → String

data FileHandle = FileHandle ( )
data IOMode = ReadMode | WriteMode | AppendMode
__openFile : FilePath → IOMode → FileHandle
__putFileStr : FileHandle → String → ( )
__readFileChar : FileHandle → Char
__readFileLine : FileHandle → String
__isEOF : FileHandle → Bool
__closeFile : FileHandle → ( )

(++) : String → String → String
show : ∀ a . a → String

readUnit : String → ( )
readBool : String → Bool
readInt : String → Int
readChar : String → Char
```

Finally, we'll be presenting the transformation of interactions with channels into calls to IO primitives. The simplest way to show this is by presenting the `__runWriteFile` function, which is called upon when the programmer calls the function `openWriteFile`.

```
__runWriteFile : FileHandle → Dual OutStream 1 → ( )
__runWriteFile fh ch =
  match ch with {
    PutChar ch →
      let ( c , ch ) = receive ch in
      __putFileStr fh ( show @Char c ) ;
      __runWriteFile fh ch ,
    PutStr ch →
      let ( s , ch ) = receive ch in
      __putFileStr fh s ;
      __runWriteFile fh ch ,
```

```
    PutStrLn ch →
      let ( s , ch ) = receive ch in
      __putFileStr fh ( s ++ "\n" ) ;
      __runWriteFile fh ch ,
    Close ch →
      __closeFile fh ; close ch
}
```

The examples previously presented represent operations directly over channels. However, in our language, there is no support for these types of operations. The possible translation of these functions to our language is by using processes instead of channels. So, for example, in the `hPutStr` function, where its signature is $Char \rightarrow OutStream \rightarrow OutStream$, instead of using the `OutStream` channel, we would use processes. The translated signature would be: $Char \rightarrow P \rightarrow P$, where P is the alternative to channels.

3

Language

Our language is a session-typed functional programming language in the manner of the SILL [15, 20, 21], which are session-typed languages based on the propositions-as-types interpretation of intuitionistic linear logic [3]. The language consists of building various modules with several concurrent or non-concurrent functions, with the purpose of solving existing problems. This section regards the previous existing language, while Section 4 regards the new addition of modules to the language. Our language combines a standard functional language, in the form of functional types, with channel-based concurrency, in the form of session types. In our language, functional types consist of a set of terms, called *expressions*, that by themselves can only be used sequentially. The session types consist of a set of terms, called *processes*, that freely use terms of the *functional* layer to create concurrent functions, which communicate on a channel-based approach. The functional layer can only depend on terms from the *process* layer if it maintains the soundness and correctness of the overall language.

3.1 Syntax

The syntax of our language is given in Figure 3.1 and defined by expressions and processes. As mentioned previously, our language ranges over functional terms using M, N and over process terms using P, Q. The functional terms define expressions like definitions of functions (sequential or recursive), variables and conditions, whereas the process terms define the concurrency aspect of the language, ranging over the definition of *sending*, *receiving* and *forwarding* other expressions, processes or channels, *spawning* and *closing* communication channels, conditions, among others. A program in our language is formed by functions and a `main` function, where we use the functions previously created. The functions are composed by a sequence of operations of the form $let\ x : T = N$, where x is of type T and can occur in N, which allows for the existence of recursive function definitions. We also allow session type declarations.

Regarding the functional terms, a standard functional language is formed with the following constructs: *basic values* (numbers, booleans, strings) and their *operators* (arithmetic,

M, N ::= V | fun $\overline{x_i} \rightarrow$ M end | let x = M in N | M N
| if M then N_1 else N_2 endif | c $\leftarrow$ {P} $\leftarrow \overline{d}$
P, Q ::= send c M ; P | send c d ; P | x : T $\leftarrow$ recv c ; P | close c
| wait c; P | fwd d c | c $\leftarrow$ spawn M d; P | case c of $\overline{l_j : (P_j)}$
| c.l; P | if M then P else Q endif | print M; P

Figure 3.1: Syntax of Expressions and Processes

relational, etc.) are abstracted by the meta-variable V; *potentially recursive let-bindings* are written in the form of $let\ x = M\ in\ N$; *function application* is noted as $M\ N$; *multi-argument λ-abstractions* have the form $fun\ \overline{x}\ \rightarrow\ M\ end$. We highlight the construct $c \leftarrow \{P\} \leftarrow \overline{d}$ which allows us to *use* a process P as a functional value, where P offers session behaviour on channel c, while using channel $\overline{d}$.

The process constructs define the session behaviour usage on the appropriate channels. In the case of $send\ c\ M;\ P$, we are sending a term M through the channel c, with the behaviour given by process P. Paired with the *send* action, $x : T \leftarrow recv\ c;\ P$ denotes an input action on channel c, receiving the value (of type T), binding it to x in the continuation P. The construct $send\ c\ d;\ P$ differs from the previous send construct in the sense that, instead of sending a term M through channel c, we are sending another channel through channel c. The process construct $fwd\ c\ d$ designates a *forwarding* interaction of contents from channel c to channel d, redirecting all inputs and outputs of c to channel d, and from channel d to c as well. Both channels must be of the same type. The construct $wait\ c;\ P$ *waits* for the closure of the session channel c, whereas the construct $close\ c$ signals the *termination* of all communication through channel c.

Our session-based communication also allows for *selection* and *branching* constructs, such as the construct $c.l;\ P$, which signals the selection of the branch labelled by l on channel c, with continuation P. Paired with this construct, the construct $case\ c\ of\ \overline{l_j : (P_j)}$ denotes an external choice on channel c, and so this process will receive on channel c some label l_i and continue according to the continuation process P_i (all possible branching options are covered by typing).

Lastly, we highlight the construct $c \leftarrow spawn\ M\ \overline{d};\ P$ which is the *process-level* dual of the *functional-level* contruct $c \leftarrow \{P\} \leftarrow \overline{d}$. The functional-level construct allows for the usage of processes as values in the functional layer, while the process-level construct allows for the usage of processes as values in other processes. It is guaranteed by type safety that the execution of process $c \leftarrow spawn\ M\ \overline{e};\ P$ will evaluate M to an embedded process of the form $c \leftarrow \{Q\} \rightarrow \overline{d}$, where channel $\overline{e}$ denotes a list of channels that will instantiate the channels $\overline{d}$ in Q, and channel c is bound in the continuation P. This way, processes P and Q will execute in parallel, each with their own channels and channel c for inter-process communication.

We will now present an example of a recursive process. Recursive processes are a combination of recursive functions, spawns and forwardings. For this recursive process,

we first define a recursive session type named `IntStream` (**line 1**), which represents the behaviour of outputting an infinite stream of integers. The `nats` function receives an integer n and produces a process that offers an `IntStream` on channel c. The function sends the integer n through channel c (**line 3**) and spawns a recursive call of `nats` $(n + 1)$ through channel d (**line 4**). Then, we connect the two channels by using the forward construct $fwd\ d\ c$ (**line 5**). The following code is the representation of the recursive function `nats`:

```
1 stype IntStream = rec x. int ∧ x;
2 let nats : int → {IntStream} =
3 fun n → c ← { send c n;
4               d ← spawn (nats (n+1));
5               fwd d c } end;
6 let main : {IntStream} = (nats 0);
```

This next example represents a Fibonacci function, which calculates the next number in the Fibonacci sequence. First, we define the recursive type `IntCStream` (**line 1**) that allows for the choice of requesting the next number in the sequence, with the keyword "`next`" (**line 5**), or to end the communication, with the use of the keyword "`stop`" (**line 9**). Our `fib` function is defined in the second line of the code presented below. It takes two arguments: an integer n, which is the *current* number of the sequence, and an integer acc, which is an *accumulator*, and it represents the *next* number in the sequence, and outputs a `IntCStream` through the channel c. In the case that the user requests the next number, first, the Integer n is sent through channel c (**line 6**), then we spawn a new recursive process of $fib\ acc\ (n + acc)$, linked to channel d (**line 7**), to calculate the next number of the sequence. This new channel d is then connected to channel c via the forward construct (**line 8**). In the case that the user requests to stop the calculation of values from the Fibonacci sequence, then we close channel c (**line 10**) and all communication ceases.

```
1  stype IntCStream rec x. &{next: int ∧ x, stop: @};
2  let fib : int → int → {IntCStream}
3  fun n acc → c ← {
4      case c of
5          next: (
6              send c n;
7              d ← spawn (fib acc (n + acc));
8              fwd d c
9          ) stop: (
10              close c
11         )
12 } end;
```

Below we present the main function that utilizes the `fib` function presented previously, requesting the first three numbers of the Fibonacci sequence.

```
1 let main : {1} =
2    f ← spawn (fib 0 1);
```

```
3     f.next;
4     x1:int ← recv f;
5     print x1;
6     f.next;
7     x2:int ← recv f;
8     print x2;
9     f.next;
10    x3:int ← recv f;
11    print x3;
12    f.stop;
13    wait f;
14    close m
15 ;;
```

We first spawn a new process associated to channel f (**line 2**), and then request the next three values of the Fibonacci sequence. The label `next` is sent through channel f (**line 3**) and we receive the result through that same channel (**line 4**). To terminate all operations, we use the `stop` label (**line 12**) and then wait for the channel to close (**line 13**).

3.2 Types

In Figure 3.2, we present the syntax of types of our language, as per [21]. Our language distinguishes between functional types, ranging over them using T and S, session types, ranging over them using A and B, and type declarations that are used for convenience, ranging over them using D. The functional types type over functional terms and those types are mostly standard, including function types $T \to S$ as well as *fundamental* data types comprised of `booleans` and `numeric` types. We highlight the type $\{\overline{B} \vdash A\}$, which is a functional term of the form $c \leftarrow \{P\} \leftarrow \overline{d}$. Process P will provide session type A on channel c by utilizing channels $\overline{d}$ in accordance with types $\overline{B}$.

The session types in our language have an important role in providing a formal foundation for verification, as well as a basic method for precisely and methodically describing complicated interaction behaviors at a high level of abstraction. The following session types describe the flow of communication operations that occur on channels: the type $A \otimes B$ represents a channel in which the process that offers data to it *sends a channel* of type A and behaves according to type B; dually, the type $A \multimap B$ represents a channel in which the process that offers data to it expects to *receive a channel* of type A and then behaves according to session type B; the types $T \wedge B$ and $T \supset B$ represent the `output` and `input`, respectively, of functional values of type T, with the continuation of type B; the session type $\oplus(\overline{l_i : A_i})$ represents a channel in which it *sends one of the labels* l_i and offers the behaviour prescribed by the session type A_i; dually, the type $\&(\overline{li : Ai})$ represents a channel in which the process that offers data to it will *receive one of the labels* l_i and then behave as session type A_i; the type variables X and the type $\mu X.A$ allow for *recursive session types*; types N may be used accordingly; type 1 represents the *inactive* session.

D ::= stype N = A
T, S ::= Num | Bool | . . . | T → S | $\{\overline{B} \vdash A\}$
A, B ::= A ⊸ B | A ⊗ B | T ∧ B | T ⊃ B | $\&\{\overline{l_i : A_i}\}$ | $\oplus\{\overline{l_i : A_i}\}$ | μX.A | X | N | 1

Figure 3.2: Functional Types (T, S), Session Types (A, B) and Declarations (D)

3.2.1 Typing Judgements

To ensure our language isn't ill-typed, we use two typing judgements: $\Psi \Vdash M : T$ and $\Psi; \Delta \vdash P :: c : A$. The first typing judgement is directed to the functional terms, in which M is of type T under the typing assumptions Ψ of the form $x : T$. The second typing judgement is directed to the process terms, where process P uses the session behaviors specified in the linear context Δ to offer session types A over channel c, under the functional typing assumptions Ψ.

To explain how the typing rules work in our language, we'll present only the key typing rules for certain constructs. First, we'll showcase the typing rule for embedding processes in functional values:

$$\frac{\Psi; \overline{d : B} \vdash P :: c : A}{\Psi \Vdash c \leftarrow \{P\} \leftarrow \overline{d} : \{\overline{B} \vdash A\}} (\{\} - I)$$

The rule presented above states that if process P *uses* channels $\overline{d}$ of type $\overline{B}$ and sends type A through channel c, then the functional term $c \leftarrow \{P\} \leftarrow \overline{d}$ is of type $\{\overline{B} \vdash A\}$, in the context where P is well-typed. The use of $\overline{d : B}$ simbolizes a set of channels d defined by a set of types B and this semantic is used throughout the explanation of our language. In contrast with this functional term, we can describe the process term that *accepts* other processes by the following rule:

$$\frac{\Psi \Vdash M : \{\overline{B} \vdash A\} \qquad \Delta' = \overline{d : B} \qquad \Psi;\ \Delta,\ c : A \vdash P :: f : C}{\Psi;\ \Delta, \Delta' \vdash c \leftarrow \textbf{spawn}\ M\ \overline{d};\ P :: f : C} (\{\} - E)$$

By initially typing M as a process value of type $\{\overline{B} \vdash A\}$, the typing rule above captures the *spawning* of the process value M in parallel with process P. To spawn this process, we need to provide it with sessions of types $\overline{B}$, and in order to satisfy this constraint we require offering the available channels $\overline{d}$ of the required types to be consumed by the process. These channels are used in the context region Δ', which satisfies the constraint and is no longer accessible to P. Afterwards, the process P communicates with process value M through channel c, of type A.

To create typing rules for process terms, we establish how to *use* and *offer* sessions at given types. The *channels used* are present in the context Δ, while the *offered channels* appear on the right-hand side, more often represented by channel c.

$$\frac{\Psi \Vdash M : T \qquad \Psi; \Delta \vdash P :: c : A}{\Psi;\ \Delta \vdash \textbf{send}\ c\ M\ ;\ P :: c : T \wedge A} (\wedge - R) \qquad \frac{\Psi,\ x : T;\ \Delta,\ c : A \vdash P :: d : D}{\Psi;\ \Delta,\ c : T \wedge A \vdash x \leftarrow \textbf{recv}\ c\ ;\ P :: d : D} (\wedge - L)$$

The *sending* and *receiving* of values is typed by the rules presented above. The typing rule on the left-hand side types a process term that *sends* a term M of type T through channel c of type $T \wedge A$ and then offers the behavior A along c. In the typing rule on the right-hand side we *receive* a value of type T bound to x, and continue as P, with the possibility to continue using the channel with type A. The next two rules follow the *right* rules and the *left* rules. The right rules define the typing of processes that *offer* sessions, while the left rules define the typing of processes that *use* sessions of a given type.

$$\frac{\Psi;\ \Delta \vdash P_1\ ::\ c : A_1 \quad \ldots \quad \Psi;\ \Delta \vdash P_n\ ::\ c : A_n}{\Psi;\ \Delta \vdash \textbf{case}\ c\ \textbf{of}\ \overline{l_j : (Pj)}\ ::\ c : \&\{\overline{l_j : A_j}\}}\ (\&-R) \qquad \frac{\Psi;\ \Delta, c : A_i \vdash P\ ::\ d : D}{\Psi;\ \Delta, c : \&\{\overline{l_j : A_j}\} \vdash c.l_i;\ P\ ::\ d : D}\ (\&-L)$$

Both rules presented above represent the processes that symbolize the use of choice, by offering a session type $\&\{\overline{l_j : Aj}\}$ on channel c with different alternatives l_i, each with a choice label with the appropriate type A_i. The left rule receives a label through channel c and uses that label to check which choice was made. The right rule involves presenting all of the different choices and offering those choices through channel c. The *fowarding* construct is a special case in our language where we use channel d of type A to offer the behavior of its type along channel c by redirecting messages sent through d to channel c and vice-versa:

$$\frac{}{\Psi;\ d : A \vdash \textbf{fwd}\ d\ c\ ::\ c : A}\ (FWD)$$

3.3 Compiler

It's important to understand how the compiler of our session-typed language works because the module system we pretend to implement must be compatible with this compiler. The compiler was implemented by Geraldo in previous work [7, 8]. Rather than having to install a potentially less efficient concurrent runtime, they decided to compile our language to Go in order to benefit from the *goroutines*, which are Go's efficient, channel-based concurrency primitives and lightweight threads. Since the Go compiler does not readily enable compiler extensions, we can take advantage of the compiler optimizations of the Go compiler by compiling to a high-level language that is comparable to our source language, sparing us the effort of implementing these strategies from the ground up.

The majority of the functions and procedures are simple to compile. Go functions are directly compiled from functions. A Go function that takes the channels d and c as arguments is built with a process value of the type $c \leftarrow \{P\} \leftarrow d$. The compilation architecture of our language takes into account the variations in the typing discipline for channels between Go and our language. The type of values that can be transferred across a channel in Go is encoded by the channel's type, and this type is fixed at the channel creation. In our session-typed language, a channel's payload is dynamic as the session protocol is

executed. In the case of channel c of type $int \wedge bool \wedge 1$, as in $send\, c\, 4; send\, c\, true; close\, c$, the payload is initially an integer, then a boolean value, and finally a termination message. In Go, a simple channel type cannot be used to directly implement this kind of pattern. In order to resolve this problem, we encode a single session channel using several Go channels, drawing inspiration from the translation of session types into linear types [4], and make use of the type information that is gathered and included into the Abstract Syntax Type during the type verification stage. To be more exact, we employ a separate Go channel for each session operation, meaning that each one entails transmitting both the payload that has to be sent and the channel that will be used for the subsequent session protocol action. As a result, we make use of Go's static typing rather than depending on runtime type casting, which has a cost. We encode each session type used in a program into a series of protocols or session type states that are represented using custom Go types. This is because session types are intrinsically stateful objects, where each communication action progresses the session to a new state.

When exchanging data, the custom type holds details about the type name, the type of data being transferred, and the state machine's following equivalent state. When it comes to a choice session type, the correspondent custom type has a mapping of labels to corresponding next states rather than merely information about the previous kind. There isn't a subsequent type for a terminal session type because communication has ended.

Initialization functions for the various types and transition procedures to move the session along are associated with each type. A sample of the code created for the initial state, represented by the type $int \wedge bool \wedge 1$, is as follows:

```
type _send_int struct { c chan int ; next *_send_bool }
func init_send_int() *_send_int { return
    &_send_int{make(chan int), nil} }
func (x *_send_int) Send(v int) *_send_bool { ... }
func (x *_send_int) Recv() (int, *_send_bool) { ... }
```

The Go struct type *_send_int* represents the first protocol state. It has a channel for exchanging integer values and a reference to the subsequent protocol state, whose type is aptly termed *_send_bool*. *Send* and *Recv* methods are linked to the session type since it indicates a value communication. They carry out the necessary communication on the underlying Go channel and return the appropriate state in response. Since state initialization is done erratically, each $init$ function only initializes the outer state; Send and Recv actions initialize the subsequent states as needed. This avoids the needless establishment of channels and offers a more straightforward initialization when recursive types are present.

It is complicated to represent recursive session types such as $rec\, x.\&\{next : int \wedge x, stop : 1\}$, seen in the type $IntCStream$ present in Section **??**. This type codifies a protocol that starts with the exchange of a message label. The protocol ends if the label is $stop$; it repeats if the label is $next$ and an integer value is transferred. As previously indicated, we use a message-label map to construct a choice session, with each label

corresponding to a suitably initialized representation of the respective branch type:

```
type _choice struct {c chan string; ls map[string]interface{}}
func init_choice() *_choice {
    m := make(map[string]interface{})
    m["next"] = init_send_int(); m["stop"] = init_close();
     return &_choice{make(chan string), m} }
```

The protocol can then be repeated because the Go type that corresponds to the next branch has an indirect reference to the choice type:

```
type _send_int struct { c chan int ; next *_choice }
```

The session type of channels c and d, of equal types, directs the compilation of the channel forwarding construct $fwd\ d\ c$. For example, a forwarder between an offered channel c, typed as $d : int \wedge 1 \vdash fwd\ d\ c :: c : int \wedge 1$ and an ambient channel d, typed as $int \wedge 1$ must accept an integer from d, send it via c, wait for d to terminate, and then end the session on c. As a result, by examining the forwarded type, the forwarder primitive compiler creates a process expression. Messages are redirected from the ambient channel to the offered channel by this mechanism, and vice versa. In the event that the type being sent is recursive, the forwarder function is merely incorporated into an infinite loop, used with caution to ensure proper variable usage throughout loop iterations and termination upon loop break.

In the definition of `fib` from Section **??** produced by our compiler, we highlight the recursive *next* branch from the forwarder, where Go variables c and d are used in accordance with the designated type and then restored with the correct values prior to the subsequent iteration:

```
for { dc1 := c.Recv(); d.Send(dc1)
  switch dc1 { case "next": d0 := d.ls["next"].(*_state_next)
               c1 := c.ls["next"].(*_state_next)
               c1d0, c1_d0 := d0.Recv()
               d = c1_d0
               c = c2.Send(c2d0)
          case "stop": ... } }
```

4

My Contribution

In this chapter, we outline the changes and additions to the grammar (4.1), parser (4.2), desugar (4.3), typechecker (4.4), compiler (4.5) and main (4.6). We will also provide some examples, namely of the modules and their corresponding header files, as well as the generated `Go` files for those modules. Finally, we will discuss the implementation of a standard library (4.7).

4.1 Grammar

The module system allows the user to modularize their code like with the use of classes in modern programming languages. New constructs were made, and two new types were created as a result: the type `imprt`, which stands for an import statement, and the type `modl`, which stands for a module. These types were added to the abstract syntax and grammar in order to support the new module system.

The first definition included in the grammar of the language was the type `modl`. This type is a representation of a module, and each file is considered a module. Therefore, if there is a file named `a.prog`, then the module that represents that file will be named `a`.

The second definition added to the grammar of the language was the type `imprt`. This type is a representation of the imported functions of a module. The module being imported should already be defined and should also have the provided functions.

After the skeleton of the modules has been set up, the next step was to implement the required tokens to enable the modules at the lexer level, and to modify the structure of the program at the parser level. To begin with, the tokens added to the lexer were `module`, `where` and `import`. In the parser, the added structure was superimposed on the existing program. Thus, by means of the existing program, the added structure was the following:

```
main:
  | MODULE m = VAR WHERE i = imprt* p = prog EOF { Modl (m, i, p) }

imprt:
  | IMPORT i = VAR L_PAR lvar = VAR+ R_PAR END { Imprt (i, lvar) }
```

Considering the above additions to the syntax, we can now attempt to provide an example of a module. Figure 4.1 illustrates a module called `a`, which is stored in the file `a.prog`, and contains an import for a function from module `b` labeled `plus_one`. The `plus_two` function of module `a` is implemented by using the imported function `plus_one` from module `b`.

```
module a where
import b (plus_one) end

plus_two : int -> {int ^ @}
fun n -> c <- {
  d <- spawn (plus_one (n+1));
  fwd d c
}
end;

;; m <- {
  d <- spawn (plus_two 1);
  a:int <- recv d;
  print a;
  wait d;
  close m
}
;;
```

Figure 4.1: Example of a module

Module `a` contains a function named `plus_two`, whose purpose is to add two to the given argument. This function is implemented by invoking the function `plus_one` from the imported module `b`. The main section of the module contains a call to the function `plus_two` with argument 1, and the result of this operation is then printed.

4.2 Parsing

The first stage is to analyze the modules for any possible syntax errors which may exist. For this task, the dune project needed to be modified to adapt the use of `menhir` to report errors [17]. Menhir is a tool that helps in error reporting while checking for grammar errors. By running the command `menhir -list-errors parser.mly`, menhir produces an output detailing all the sentences that can be formed, which can be classified as reaching all error states. This output is then placed into a file called `errorMessages.messages`. In this instance, the output went to that file and the proper error messages were set, and in the dune file located in the `lib` package, there is a section which takes care of these files. The contents of the `errorMessages.messages` file should be correct (that is, every sentence causes an error on its last token), irredundant (no two sentences lead to the same error state), and complete (every error state is reached by some sentence). In

this dune file, first, we perform the completeness check and we use the output of that operation to perform the correctness and irredundancy check. Completeness check is done by using the command `menhir -compare-errors errorMessages.auto.messages -compare-errors errorMessages.messages` to compare the two `.messages` files. The output of this operation is afterwards used in the correctness and irredundancy check. By using the command `menhir -compile-errors errorMessages.messages`, menhir compiles the `.messages` file to an Ocaml function which is written to the file specified.

We first obtain the complete list of modules, including imported modules, in order to begin parsing. Then, we iterate through this list, parsing each file, searching for any syntax errors.

4.3 Desugaring

The next step after parsing the module is desugaring it. To do this, we must extend the `syntax.ml` file. This phase converts `InputSyntax` types into language types that will be used by the rest of the program. To be more precise, we desugar the modules in the list of modules that need to be compiled one at a time, and after this operation has ended, we use the output for the typechecking phase.

4.4 Typechecking

The typechecking phase is one of the most important phases, as it verifies if there is an error with any of the imports. The most important aspects of this phase are the verification of cycles within the imports (Section 4.4.1), and the generation of header files, which act as signature files to check if a module has already been compiled (Section 4.4.2). The typechecking phase starts by checking if there are any cycles in the imports, and then typechecks the contents of each module. During this process, to allow the modules to use the imported functions, the typechecker adds the imported functions to the module that imported them. This way, when compiling the module, the imported functions are already present in the module.

4.4.1 Circular Imports

A circular import occurs when two modules import each other, creating a cycle in the import graph. The problems created by having cycles in the import graph would cause the program to enter an infinite loop of calls to each imported module. To address this issue, we have implemented a verification step that detects cycles and returns the correct order in which modules should be typechecked and compiled. Performing this verification as early as possible allows us to detect circular imports early, without the need to compile the entire program. If a cycle is detected, an error occurs, the program fails and the

compilation is never reached. The following function addresses the detection of circular imports:

```
let check_circular lmodl =
try
  let comp = create_edges lmodl in
    let scc_list = kosaraju comp in
    if List.length scc_list = List.length lmodl then(
      scc_list)
    else(
      print_endline(string_from_err(error(NoCyclesAllowed)));
      raise (error (NoCyclesAllowed))
    )
with e ->
  print_endline("A circular import has been found: "^Printexc.to_string e);
  raise e
```

This verification starts by creating an edge graph of the modules with the function `create_edges`. Then, we use Kosaraju's algorithm to detect every Strongly Connected Component (SCC) in the graph. A SCC is a subset of the edge graph, in which each node in this subgraph is able to reach every other node in the subgraph, and it can also be reached by every other node. Kosaraju's algorithm is used to detect these SCCs. This algorithm is an efficient way to detect SCCs in a graph, as it only requires two passages of the edge graph. This algorithm consists of three steps: first, it does a depth-first search on the graph; then, it transposes the graph given by the first search; finally, it does a depth-first search on the transposed graph. The first search is used to determine and record the order in which the nodes are visited, while the second search processes the nodes in order of decreasing finish time and creates the list of SCCs. After running the edge graph through our algorithm, we obtain the list of SCCs and check if the number of SCCs found is equal to the number of modules in the graph. If this is true, then the list of SCCs with the correct order is returned. This means that each SCC is comprised of a single module, and therefore there are no circular imports and the verification is successful. Otherwise, an error is raised. The following example is used to explain how the verification works.

```
module a where
import b (plus_one) end
plus_two : int -> {int ^ @}
fun n -> c <- {
    d <- spawn(plus_one(n+1));
    fwd c d
}
end;

;; m <- {
    d <- spawn (plus_two 1);
    a:int <- recv d;
    print a;
    wait d;
    close m
};;
```

```
module b where
plus_one : int -> {int ^ @}
fun n -> c <- {
    send c (n+1);
    close c
}
end;

;; m <- {
  d <- spawn (plus_one 1);
  a:int <- recv d;
  print a;
  wait d;
  close m
};;
```

The module given on the left is a module that imports the function `plus_one` of the module on the right. This function is used to help the calculation of the function `plus_two`. In this case, module a imports module b and module b does not import anything. Therefore, the list of SCCs would be [[b], [a]]. Since the size of the list of modules is the same as the size of the list of SCCs, there is no error during the verification of cycles. However, in the case that module b imports module a, the list of SCCs would be [[b, a]], and since the size of this list is different than the size of the module list, a cycle would be found and an error would occur.

4.4.2 Header Files

As previously stated in Section 4.4, the generation of header files is a crucial part of the verification of already compiled modules. This feature generates a header file for each module, containing the signatures of the functions defined in the module, as well as the signatures of the functions imported from other modules. It is possible to implement this feature in two different phases: in the typeckecking phase, and in the compilation phase. Each of these phases has its own advantages and disadvantages. By implementing it in the typechecking phase, we create the header file of a module as soon as it is typechecked. Since the file is created in the typechecking phase, we ensure that only type-safe and valid declarations are included in the header files. We chose to implement this feature in the typechecking phase, as it allows us to create the header files earlier in the process, and ensures that the interface of each module is consistent throughout the compilation process.

These files are used to verify if a module needs to be recompiled or not. In case a recompilation is needed, we simply need to check if the timestamp on the header file corresponds to the timestamp on the respective program file. If it is, then the entire module is desugared, typechecked and compiled again, otherwise we can skip the entire process for that module, which can reduce the overall compilation time of the project.

The following example represents the header file created for module `a` given in the previous example:

```
Header for module a
val plus_two : int -> proc(!int.end, [])
From module b
val plus_one : int -> proc(!int.end, [])
```

This header file contains the signature of the function `plus_two` defined in module `a`, as well as the signature of the function `plus_one` imported from module `b`.

4.5 Compiling

Previously, the compilation process compiled the program to `Go`, by creating a `.go` file and outputting the translation of the program to `Go` code into it. Now, it compiles each module into its own `Go` file, inside a package with the name of the module. It's important to note that, for the purpose of the correct use of the imported functions, the program of each module outputted by the typechecker contains the imported functions. In order to allow modules to use the imported functions, the translated `Go` code of the imported functions is added to the `Go` file of the module that imported it. This way, the module importing those functions can easily use them. After being parsed, desugared and typechecked, the module enters the compilation with the output given by the typechecker.

The following example is the `Go` file created upon the compilation of the module `a` presented in Section 4.4.1:

```
package a
import ("fmt"
)
type _state_1 struct {
  c chan interface{}
}
func init_state_1(c chan interface{}) *_state_1 { return &_state_1{ c } }
func (x *_state_1) Send(v interface{}) { x.c <- v }
func (x *_state_1) Recv() interface{} { return <-x.c }

type _state_0 struct {
  c chan interface{}
  next *_state_1
}

func init_state_0(c chan interface{}) *_state_0 { return &_state_0{ c, nil } }
func (x *_state_0) Send(v int) *_state_1 { if x.next == nil {
  x.next = init_state_1(x.c)
}; x.c <- v; return x.next}
func (x *_state_0) Recv() (int, *_state_1) { if x.next == nil {
  x.next = init_state_1(x.c)
}; return (<-x.c).(int),x.next}
```

```
func plus_one(n int) func (_x *_state_0) {
  return func (c *_state_0){
    c0 := c.Send((n + 1))
    c0.Send(nil)
  }
}
func plus_two(n int) func (_x *_state_0) {
  return func (c *_state_0){
    d := init_state_0(make(chan interface{}))
    go plus_one((n + 1))(d)
    // FWD c d Start
    cd, d0 := d.Recv()
    c0 := c.Send(cd)
    d0.Recv()
    c0.Send(nil)
    return
    // FWD c d End
  }
}
func main () {
  m := init_state_1(make (chan interface{}))
  go func () {
    m.Recv()
  }()
  func (m *_state_1){
    d := init_state_0(make(chan interface{}))
    go plus_two(1)(d)
    a, d0 := d.Recv()
    fmt.Printf("%v\n",a)
    d0.Recv()
    m.Send(nil)
  }(m)
}
```

As previously explained, the compiled file has the imported function `plus_one`, and because it was added to the file that imported it, it can be used inside of that module.

4.6 Main

The final step of the implementation is integrating previous additions and alterations to the parsing, desugaring, typechecking and compiling phases into the entry file `main.ml`. First, we retrieve the entire list of reachable modules from the input files. With this list formed, we parse this list to check if there are any syntax errors in any of those files (Section 4.2). Then, we verify what files have been previously compiled and remove them from the list of modules that need to be typechecked and compiled. This verification is one of the most important aspects of this file, because it allows us to skip the typechecking and compilation of modules that have already been compiled. The typechecking phase of the modules

generates the header files corresponding to each module, and by comparing the timestamp of the header file with the timestamp of the corresponding module, we can determine if the module has been previously compiled. If it has, we can skip its recompilation (Section 4.4.2). After this verification, we typecheck (Section 4.4) and compile (Section 4.5) the remaining modules.

4.7 Standard Library

The standard library is an important part of the language, as it provides pre-defined modules containing operations on booleans (Section 4.7.1), integers (Section 4.7.2) and data structures (Section 4.7.3). Each of these modules follows the rules of the language, and by having these modules, the user has access to a wide range of already implemented functions usable in their code.

4.7.1 BooleanUtils

The `booleanUtils` package provides two modules with sets of operations on boolean values. The two modules in this package are: `boolOps` and `boolCompare`. The `boolOps` file performs operations on boolean values, such as `notOp`, `andOp`, `orOp` and `xorOp`. The `notOp` function receives a boolean and negates it. The `andOp`, `orOp` and `xorOp` functions perform, respectively, the logical `AND`, `OR` and `XOR` operations on the given input values. The Figure 4.2 shows the implementation of this module:

The `boolCompare` file performs comparissons between boolean values, such as `isTrue` and `isFalse`. The function `isTrue` returns true if the given boolean is also true, while the function `isFalse` returns true if the given boolean is false. The Figure 4.3 shows the implementation of this module:

4.7.2 MathUtils

The `matUtils` package provides two modules with sets of operations on integer values. The two modules in this package are: `intOps` and `intCompare`. The `intOps` file performs operations on integer values, such as `fib`, `pow`, `abs` and `sqrt`. The `fib` function calculates the next value of the Fibonacci sequence based on the two given values. The `pow` function calculates the power of the first given value to the second given value. The `abs` function returns the absolute value of the given integer value. The `sqrt` function calculates the square root of the given integer value with the help of an auxiliary function `subtract`. The Figure 4.4 shows the implementation of this module:

```
module intOps where
stype IntCStream rec x. &{next: int^x, stop: @};
fib : int -> int -> {IntCStream}
fun a b ->
  c <- {
    case c of
```

```
module boolOps where
notOp : bool -> {bool ^ @}
fun b -> c <- {
    send c (not b);
    close c
} end;
andOp : bool -> bool -> {bool ^ @}
fun a b -> c <- {
    if a then
        send c b;
        close c
    else
        send c false;
        close c
} end;
orOp : bool -> bool -> {bool ^ @}
fun a b -> c <- {
    if a then
        send c true;
        close c
    else
        send c b;
        close c
} end;
xorOp : bool -> bool -> {bool ^ @}
fun a b -> c <- {
    if a = b then
        send c false;
        close c
    else
        send c true;
        close c
} end;
;;
m <- {
    close m
};;
```

Figure 4.2: Module `boolOps`

```
    next: (
      send c b;
      d <- spawn (fib b (a + b));
      fwd c d
    ) stop: (
        close c
    )
} end;
fact : int -> {int ^ @}
fun n -> c <- {
  if n > 1 then
    d <- spawn (fact(n-1));
    fac:int <- recv d;
    wait d;
    send c (n*fac);
    close c
```

```
module boolCompare where
isTrue : bool -> {bool ^ @}
fun a -> c <- {
    if a then
        send c true;
        close c
    else
        send c false;
        close c
} end;
isFalse : bool -> {bool ^ @}
fun a -> c <- {
    if a then
        send c false;
        close c
    else
        send c true;
        close c
} end;

;;
m <- {
    close m
};;
```

Figure 4.3: Module `boolCompare`

```
  else
    send c 1;
    close c
} end;
pow : int -> int -> {int ^ @}
fun a b -> c <- {
  if (b = 0) then
      send c 1;
      close c
  else
    d <- spawn (pow a (b-1));
    res:int <- recv d;
    send c (a * res);
    wait d;
    close c
} end;
abs : int -> {int ^ @}
fun n -> c <- {
  if (n < 0) then
    send c (0-n);
    close c
  else
    send c n;
    close c
} end;
sqrt : int -> {int ^ @}
```

```
fun n -> c <- {
  if n < 0 then
    send c -1;
    close c
  else
    d <- spawn (subtract 0 n);
    res:int <- recv d;
    send c res;
    wait d;
    close c
} end;
;; m <- {
    close m
};;
```

Figure 4.4: Module `intOps`

The `intCompare` file performs comparissons between integer values, such as `isEven`, `isOdd`, `isBiggerThan`, `isSmallerThan`, `isPositive`, `isNegative`, `max`, `min` and `isPrime`. The `isEven` function returns true if the given integer value is even, while the `isOdd` function does the opposite. The functions `isBiggerThan` and `isSmallerThan` compare two integer values and return true if the first given value is bigger or smaller than the second given value, respectively. The `isPositive` and `isNegative` functions return true if the given integer value is positive or negative, respectively. The `max` and `min` return the maximum and minimum values between two given integer values, respectively. The `isPrime` function returns true if the given integer value is a prime number. This function has two auxiliary functions: `subtractUntilZero` and `isDivisible`, which are not present in the figure below. The first function receives two integer values and recursively subtracts the second value from the first until the result is either zero or negative. The second function calculates if the first given integer value is divisible by the second given integer value. If it is, then it returns true, otherwise it returns false. The Figure 4.5 shows the implementation of this module:

```
module intCompare where
isEven : int -> {bool ^ @}
fun a -> c <- {
  if (a = 0) then
    send c true;
    close c
  else
    if (a = 1) then
      send c false;
      close c
    else
      d <- spawn (isEven (a - 2));
      res:bool <- recv d;
      wait d;
```

```
      send c res;
      close c
} end;
isOdd : int -> {bool ^ @}
fun a -> c <- {
  if (a = 0) then
    send c false;
    close c
    else
      if (a = 1) then
        send c true;
        close c
      else
        d <- spawn (isOdd (a - 2));
        res:bool <- recv d;
        wait d;
        send c res;
        close c
} end;
isBiggerThan : int -> int -> {bool ^ @}
fun a b -> c <- {
  if a > b then
    send c true;
    close c
  else
    send c false;
    close c
} end;
isSmallerThan : int -> int -> {bool ^ @}
fun a b -> c <- {
  if a > b then
    send c false;
    close c
  else
    send c true;
    close c
} end;
isPositive : int -> {bool ^ @}
fun a -> c <- {
  if a > -1 then
    send c true;
    close c
  else
    send c false;
    close c
} end;
isNegative : int -> {bool ^ @}
fun a -> c <- {
  if a < 0 then
    send c true;
```

```
    close c
  else
    send c false;
    close c
} end;
isPrime : int -> {bool ^ @}
fun a -> c <- {
  if (a < 2) then
    send c false;
    close c
  else
    d <- spawn (isDivisible a 2);
    res:bool <- recv d;
    send c (not res);
    wait d;
    close c
} end;
max : int -> int -> {int ^ @}
fun a b -> c <- {
  if a > b then
    send c a;
    close c
  else
    send c b;
    close c
} end;
min : int -> int -> {int ^ @}
fun a b -> c <- {
  if a > b then
    send c b;
    close c
  else
    send c a;
    close c
} end;
;;
m <- {
  close m
};;
```

Figure 4.5: Module `intCompare`

4.7.3 DataStructures

The `dataStructures` package provides three modules with different data structures available to be imported and used in the user's code. The three modules in this package are: `intQueue`, `intList` and `intStack`. The `intQueue` file contains functions for working with a concurrent integer queue, using session types. The queue is represented by a

session type with two operations: `enq` and `deq`. The `enq` operation enqueues a new integer element into the queue, while the `deq` dequeues an integer element from the queue. This module has three functions: `elem`, `empty` and `deallocQueue`. The `elem` function adds or removes an element from the queue, the `empty` function creates an empty queue, and the `dealloc` function removes all elements from the queue. The Figure 4.6 shows the implementation of this module:

```
module intQueue where
stype Queue rec x. &{enq: int => x, deq: +{some: int ^ x, none: @}};
elem : int -> {Queue <- r:Queue}
fun x ->
        q <- {
    case q of
                enq:(
                        y:int <- recv q;
                        r.enq; send r y;
        q_ <- spawn (elem x) r;
                fwd q q_
                )
                deq:(
                        q.some; send q x;
        fwd q r
        )
        }
end;
empty : unit -> {Queue}
fun u ->
        q <- {
    case q of
                enq:(
                        y:int <- recv q;
                        print y;
        e <- spawn (empty ());
                q_ <- spawn (elem y) e;
                        fwd q q_
                )
    deq:(
                        q.none;
                        close q
                )
  }
end;
deallocQueue : unit -> {@ <- q:Queue}
fun u ->
  c <- {
                q.deq;
                case q of
                some:(
                        i:int <- recv q;
```

```
                        d <- spawn (deallocQueue ()) q;
        fwd c d
        )
                none:(
                        wait q;
                        close c
    )
        }
end;
;; c <- {
        close c
};;
```

Figure 4.6: Module `intQueue`

The `intList` file contains functions for working with a concurrent linked list of integers, using session types. The list of integers is represented by a session type with two operations: `null` and `cons`. The `null` operation represents an empty list, while the `cons` operation represents the contents of the list. This module has three base functions: `null`, `cons` and `dealloc`. The `null` function creates an empty list, The `cons` function creates an entry in the list, and the `dealloc` function removes all elements from the list. The additional functions to operate on lists allow us to retrieve the sum of all values in the list with the function `addAllInList`, to filter the list based on a given integer value with the functions `filterGreaterThan`, `filterSmallerThan` and `filterEqualTo`, and to find the maximum and minimum values in the list with the functions `findMax` and `findMin`. The last two functions use the functions `max` and `min` from the module `intCompare` from Figure 4.5. The Figure 4.7 shows the implementation of this module:

```
module intList where
import intCompare (max, min) end
stype IntList rec x. +{null: @, cons: int ^ (x) * @};
null : unit -> {IntList}
fun n ->
        c <- {
                c.null;
                close c
        }
end;
cons : int -> {IntList <- l:IntList}
fun v -> c <- {
        c.cons;
        send c v;
        print v;
        l_ <- spawn { fwd l_ l} l;
        sendc c l_;
        close c
}
end;
```

```
dealloc : unit -> {@ <- l:IntList}
fun v -> c <- {
  case l of
        null:(
                wait l;
                close c
        )
        cons:(
        n:int <- recv l;
        l_:IntList <- recvc l;
                wait l;
                d <- spawn (dealloc ()) l_;
                fwd c d
        )
}
end;
addAllInList : unit -> {int ^ @ <- l:IntList}
fun a -> c <- {
        case l of
        null: (
                wait l;
                send c 0;
                close c
        )
        cons: (
                n:int <- recv l;
                l_:IntList <- recvc l;
                wait l;
                d <- spawn (addAllInList ()) l_;
                rest_sum:int <- recv d;
                send c (n + rest_sum);
                wait d;
                close c
        )
}
end;
filterGreaterThan : int -> {IntList <- l:IntList}
fun a -> c <- {
        case l of
        null: (
                wait l;
                e <- spawn (null ());
                fwd c e
        )
        cons: (
                n:int <- recv l;
                l_:IntList <- recvc l;
                wait l;
                if n > a then
                        d <- spawn (filterGreaterThan a) l_;
```

```
                        e <- spawn (cons n) d;
                        fwd c e
                else
                        d <- spawn (filterGreaterThan a) l_;
                        fwd c d
        )
}
end;
filterSmallerThan : int -> {IntList <- l:IntList}
fun a -> c <- {
        case l of
        null: (
                wait l;
                e <- spawn (null ());
                fwd c e
        )
        cons: (
                n:int <- recv l;
                l_:IntList <- recvc l;
                wait l;
                if n < a then
                        d <- spawn (filterSmallerThan a) l_;
                        e <- spawn (cons n) d;
                        fwd c e
                else
                        d <- spawn (filterSmallerThan a) l_;
                        fwd c d
        )
}
end;
filterEqualTo : int -> {IntList <- l:IntList}
fun a -> c <- {
        case l of
        null: (
                wait l;
                e <- spawn (null ());
                fwd c e
        )
        cons: (
                n:int <- recv l;
                l_:IntList <- recvc l;
                wait l;
                if n = a then
                        d <- spawn (filterEqualTo a) l_;
                        e <- spawn (cons n) d;
                        fwd c e
                else
                        d <- spawn (filterEqualTo a) l_;
                        fwd c d
        )
```

```
}
end;
findMax : unit -> {int ^ @ <- l:IntList}
fun a -> c <- {
        case l of
        null: (
                wait l;
                send c 0;
                close c
        )
        cons: (
                n:int <- recv l;
                l_:IntList <- recvc l;
                wait l;
                d <- spawn (findMax ()) l_;
                rest:int <- recv d;
                e <- spawn (max n rest);
                aux:int <- recv e;
                wait e;
                send c aux;
                wait d;
                close c
        )
}
end;
findMin : unit -> {int ^ @ <- l:IntList}
fun a -> c <- {
        case l of
        null: (
                wait l;
                send c 0;
                close c
        )
        cons: (
                n:int <- recv l;
                l_:IntList <- recvc l;
                wait l;
                d <- spawn (findMin ()) l_;
                rest:int <- recv d;
                e <- spawn (min n rest);
                aux:int <- recv e;
                wait e;
                send c aux;
                wait d;
                close c
        )
}
end;
;; c <- {
        close c
```

```
};;
```

Figure 4.7: Module `intList`

The `intStack` file contains functions for working with a concurrent integer stack, using session types and the previously implemented `intList`. The stack of integers is represented by a session type with three operations: `push`, `pop` and `dealloc`. The `push` operation pushes a new element onto the stack, the `pop` operation pops an element from the stack, and the `dealloc` operation removes all elements from the stack. This module has one function that implements each of the operations in the session type. Since the stack is implemented using a linked list, the base functions of the `intList` module were imported (`IntList`, `null`, `cons`, `dealloc`) and used in the implementation of the stack. The Figure 4.8 shows the implementation of this module:

```
module intStack where
import intList (IntList, null, cons, dealloc) end
stype IntStack rec x. &{push: int => x, pop: +{none: unit ^ x, some: int ^ x}, dealloc: @};
stack : unit -> {IntStack <- l:IntList}
fun n ->
        c <- {
                case c of
                push: (
                        v:int <- recv c;
                        l_ <- spawn (cons v) l;
                        d <- spawn (stack ()) l_;
                        fwd c d
                )
                pop: (
                        case l of
                        null: (
                                wait l;
                                c.none;
                                send c ();
                                l_ <- spawn (null ());
                                d <- spawn (stack ()) l_;
                                fwd c d
                        )
                        cons: (
                                v:int <- recv l;
                                l_:IntList <- recvc l;
                                wait l;
                                c.some;
                                send c v;
                                d <- spawn (stack ()) l_;
                                fwd c d
                        )
                )
                dealloc: (
```

```
                        d <- spawn (dealloc ()) l;
                        fwd c d
                )
        }
end;
;; c <- {
        close c
};;
```

Figure 4.8: Module `intStack`

5

Evaluation

5.1 Modules

The module system's implementation was a key aspect of this work. We aimed to improve the maintainability, modularity and organization of the user's code. The module system enables the definition of independent modules, each of which with its unique set of imports and functions. By using this approach, we can manage dependencies more effectively and reduce the impact of changes in one module on others.

To validate the implementation of the module system, we created a set of examples that examined a series of scenarios, such as:

- A module with no imports.
- A module that imports one or more functions from another module.
- Compilation of a module that has not been compiled.
- Recompilation of a module that has already been compiled.
- Recompilation of a module after being modified.
- A set of modules with circular dependencies between them.
- A set of modules with complex dependency graphs.

The results obtained from all of these cases show that the module system can correctly handle all scenarios. In the case of a module with no imports, the system correctly generates the header file corresponding to the module, containing all of its functions, and compiles it, generating its `Go` file. The Figure 5.1 represents a module with no imports and a single function:

```
module noImprt where
plus_one : int -> {int ^ @}
fun n -> c <- {
    send c (n+1);
```

```
    close c
}
end;
;; m <- {
    d <- spawn (plus_one 1);
    a:int <- recv d;
    print a;
    wait d;
    close m
};;
```

Figure 5.1: Module with no imports.

In the case of a module that imports one or more functions from another module, the system correctly generates the header file corresponding to the module, containing the signatures of the imported functions as well. As for the compiled `Go` file, it contains all of the modules functions, as well as the imported functions. The Figure 5.2 represents a module that imports the module `noImprt` of Figure 5.1 and uses an imported function:

```
module oneImprt where
import noImprt (plus_one) end
plus_two : int -> {int ^ @}
fun n -> c <- {
    send c (plus_one (n+1));
    close c
}
end;
;; m <- {
    d <- spawn (plus_two 1);
    a:int <- recv d;
    print a;
    wait d;
    close m
};;
```

Figure 5.2: Module with one import.

In the case of the compilation of a module that has not been compiled, the output generated by the system allows us to confirm that the module was correctly compiled. The Figure 5.3 shows the output given by the system upon the compilation of the module of Figure 5.1:

```
The .progh file does not exist: noImprt.prog
Desugaring modules...
Modules have been desugared...
```

```
Typechecking modules...
Typechecking module: noImprt
Generating header file for module: noImprt.prog
Modules have been typechecked...
Compiling modules...
Compiling module: noImprt
Modules have been compiled...
```

Figure 5.3: Output given on compilation of module `noImprt`.

In the case of the recompilation of a module that has already been compiled, the output given by the system provides information about the modules being compiled. By using the module on Figure 5.1 and the output of the program, we can determine that the module was not recompiled because it had been previously compiled. The Figure 5.4 shows the output of the system upon attempting to recompile the module of Figure 5.1:

```
The file has already been compiled: noImprt.prog
Desugaring modules...
Modules have been desugared...
Typechecking modules...
Modules have been typechecked...
Compiling modules...
Modules have been compiled...
```

Figure 5.4: Output given on recompilation of module `noImprt`.

In the case of recompilation of a module after being modified, the output given by the system demonstrates that the module has been previously compiled and needs to be recompiled. By changing the module `noImprt` of Figure 5.1 and compiling it, we obtain the output given in Figure 5.5:

```
The file needs to be recompiled: noImprt.prog
Desugaring modules...
Modules have been desugared...
Typechecking modules...
Typechecking module: noImprt
Generating header file for module: noImprt.prog
Modules have been typechecked...
Compiling modules...
Compiling module: noImprt
Modules have been compiled...
```

In the case of a set of modules with circular dependencies between them, the system checks for the existence of a cycle and interrupts the entire process if a cycle is found.

Figure 5.5: Output given on recompilation of module `noImprt` after modification.

For this purpose, we will present three modules that have a circular import: module `dependency1` and module `dependency2`. The modules are as shown in Figures 5.6 and 5.7.

```
module dependency1 where
import dependency2 (plus_three) end

plus_four : int -> {int ^ @}
fun n -> c <- {
    d <- spawn (plus_three (n+1));
    fwd c d
}
end;
;; m <- {
    d <- spawn (plus_four 1);
    a:int <- recv d;
    print a;
    wait d;
    close m
};;
```

Figure 5.6: First module of the circular dependency.

```
module dependency2 where
import dependency1 (plus_four) end

plus_three : int -> {int ^ @}
fun n -> c <- {
    d <- spawn (plus_four (n-1));
    fwd c d
}
end;
;; m <- {
    d <- spawn (plus_three 1);
    a:int <- recv d;
    print a;
    wait d;
    close m
};;
```

Figure 5.7: Second module of the circular dependency.

When compiling the modules of the Figures 5.6 and 5.7, the output given by the system reveals that an error has occurred regarding the imports. The Figure 5.8 shows the output given by the system:

```
The .progh file does not exist: dependency1.prog
The .progh file does not exist: dependency2.prog
Desugaring modules...
Modules have been desugared...
Typechecking modules...
A circular import has been found: Sessint.Printer.TError(4)
Fatal error: exception Sessint.Printer.TError(4)
```

Figure 5.8: Output of the compilation of a cycle.

Finally, in the case of a set of modules with complex dependency graphs, the module of Figure 5.9 demonstrates a module that belongs to a complex dependency graph:

```
module severalImprts where
import oneImprt (plus_two) end
import intStack (IntStack, stack) end
import intQueue (Queue, elem, empty, deallocQueue) end

plus_three : int -> {int ^ @}
fun n -> c <- {
    d <- spawn (plus_two (n+1));
    fwd c d
}
end;

;; m <- {
    d <- spawn (plus_three 1);
    a:int <- recv d;
    wait d;
    b <- spawn (empty ());
    b.enq; send b a;
    b.enq; send b 3;
    q <- spawn (deallocQueue ()) b;
    wait q;
    close m
};;
```

Figure 5.9: Module `severalImprts`.

The module `severalImprts` of Figure 5.9 imports the module `oneImprt` of Figure 5.2, which then imports the module `noImprt` of Figure 5.1. The second import is of the data structure `intStack`, which is presented in the Figure 4.7.3 of the Section 4.7.3, which then imports the module `intList`, which is presented in Figure 4.7.3. The final import refers to the module `intQueue`, which is present in the Figure 4.7.3 of Section 4.7.3.

The results of this evaluation show that the module system is able to handle several modules, and it also improved readability and organization of the code.

5.2 Standard Library

The implementation of the standard library allows the language to be extended by offering the user a set of functionalities that are usually present in other programming languages. With this goal in mind, we have correctly offered the user modules to perform mathematical and boolean operations, as well as infrastructures to deal with data structures, such as lists, queues and stacks.

We evaluate the standard library based on its integration in the language and usability in the development of programs by the user. The standard library was used in its own implementation, namely of the data structure `intStack`, and of the module `severalImprts` of Figure 5.9, and the results demonstrate that it is correctly integrated into the language and can be used by the user without problems.

Moreover, the modular design of the language allows for an easier extension of the standard library. The standard library can be extended by adding new modules, without having to worry about already existing modules.

6

Conclusions

In this thesis, a module system was implemented in a session-typed language, previously developed by Geraldo [7, 8] and optimized by Faria [6]. The module system allows the user to define dependent and independent modules, each with its unique set of imports and functions. A standard library was also created, with the purpose of providing reusable functionality and assisting the user in building programs. We also developed a set of examples to demonstrate the capabilities of the module system and of the standard library.

The module system was created mainly by extending the syntax, typechecker and compiler of the language. The important functionalities that accompany the module system are the verification of cycles and the correct generation of the header files. These files are essential for the correct compilation of the modules, as they are used to verify if a module needs to be recompiled or not. Another possible implementation of the module system is an implementation similar to Haskell's module system. Haskell allows modules to import each other in different manners, such as qualified imports, hiding imports, and importing a module and using it with a different name. Additionally, Haskell also uses polymorphism to define typeclasses, which could be added as a part of the standard library.

Future work can focus on improving the use of modules and imports by using Go's own module system, improving the standard library, and adding polymorphism to the language. When compiling a module, the generated Go file should contain Go code related to its module system, which should be used to define a module. The standard library can be improved by adding functions that have special functionalities, such as I/O operations, with the use of Go's own libraries. If polymorphism is added to the language, the standard library can be further improved to include typeclasses.

Bibliography

[1] B. Almeida, A. Mordido, and V. T. Vasconcelos. "FreeST: Context-free Session Types in a Functional Language". In: *Electronic Proceedings in Theoretical Computer Science 291* (2019), pp. 12–23. DOI: 10.4204/eptcs.291.2. URL: https://doi.org/10.4204%2Feptcs.291.2 (cit. on p. 15).

[2] D. Barros. "Shared Channels on Context-free Session Types". In: *Preliminary Report for Master Thesis in Computer Science. FCUL* (2022) (cit. on p. 15).

[3] L. Caires and F. Pfenning. "Session types as intuitionistic linear propositions". In: *CONCUR*. Lecture Notes in Computer Science 6269 (2010), pp. 222–236 (cit. on p. 19).

[4] O. Dardha, E. Giachino, and D. Sangiorgi. "Session types revisited". In: *PPDP* (2012), pp. 139–150 (cit. on p. 25).

[5] *Eq Type Class*. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Eq.html (cit. on p. 8).

[6] J. Faria. "An optimizing compiler for a session-typed functional language". In: *Master Thesis in Computer Science. NOVA School of Science and Technology* (2023) (cit. on p. 53).

[7] J. Geraldo. "Making Session Types Go". In: *Master Thesis in Computer Science. NOVA School of Science and Technology* (2022-11) (cit. on pp. 1, 24, 53).

[8] J. Geraldo and B. Toninho. "Making Session Types Go - Compilation of a Session Typed Functional Language to Go". In: () (cit. on pp. 24, 53).

[9] *Haskell Programming Language*. URL: https://en.wikipedia.org/wiki/Haskell (cit. on p. 4).

[10] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[11] *Modular Programming*. URL: https://codelabs.rocks/blog/detailed-guide-for-modular-programming-concept (cit. on p. 3).

[12] *Ocaml Functors*. URL: https://ocaml.org/docs/functors (cit. on p. 13).

[13] *Ocaml Modules*. URL: https://ocaml.org/docs/modules (cit. on p. 11).

[14] *Ord Type Class*. URL: https://hackage.haskell.org/package/base-4.20.0.1/docs/Prelude.html#t:Ord (cit. on p. 8).

[15] F. Pfenning and D. Griffith. "Polarized substructural session types". In: *FoSSaCS*. Lecture Notes in Computer Science 9034 (2015), pp. 3–22 (cit. on p. 19).

[16] *Polymorphism in Haskell*. URL: https://wiki.haskell.org/Polymorphism (visited on 2015-01-21) (cit. on p. 8).

[17] F. Pottier. *Menhir Manual*. URL: https://gallium.inria.fr/~fpottier/menhir/manual.html#sec72 (visited on 2025-03-10) (cit. on p. 28).

[18] *Read Type Class*. URL: https://hackage.haskell.org/package/base-4.9.0.0/docs/Text-Read.html#t:Read (cit. on p. 8).

[19] *Show Type Class*. URL: https://hackage.haskell.org/package/base-4.20.0.1/docs/Prelude.html#v:show (cit. on p. 8).

[20] B. Toninho, L. Caires, and F. Pfenning. "Dependent session types via intuitionistic linear type theory". In: *PPDP* (2011), pp. 161–172 (cit. on p. 19).

[21] B. Toninho, L. Caires, and F. Pfenning. "Higher-Order Processes, Functions, and Sessions: A Monadic Integration". In: *Programming Languages and Systems* (2013). Ed. by M. Felleisen and P. Gardner, pp. 350–369. DOI: https://link.springer.com/chapter/10.1007/978-3-642-37036-6_20 (cit. on pp. 1, 19, 22).

[22] *Type Classes*. URL: https://www.haskell.org/tutorial/classes.html (cit. on p. 8).