UNIVERSITY OF CALGARY

Implementation of Message Passing Language

by

Prashant Kumar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

February, 2018

Abstract

Message Passage Language (MPL) is a programming language based on the work of Cockett and Pastro. MPL is a statically typed concurrent programming language with *message passing* as the concurrency primitive. It brings communication safety to interacting processes using a type system. MPL consists of two languages, concurrent MPL and sequential MPL, which can interact with each other. Concurrent MPL programs are written using concurrency constructs built into the language and protocols, which are concurrent data types. These concurrency constructs allow intuitive modelling of real world concurrency scenarios. Sequential MPL is a functional programming language. In addition to data definitions, sequential MPL allows codata definitions, which can model infinite structures. Sequential MPL allows for disciplined recursion using folds and unfolds in addition to normal recursion.

In this thesis, we develop the first prototype of a compiler for MPL. We reformulate MPL's design to allow normal recursion in addition to primitive recursion, the only form of recursion allowed in previous designs. In light of the changes made to MPL, we describe MPL's type system. Then we develop an algorithm for type inferencing MPL programs, and implement it. In addition, we develop and implement an abstract machine to run MPL programs. We also develop the intermediate languages through which MPL programs compile to the abstract machine. We describe and implement the algorithms used in the compilation of MPL programs to the abstract machines, namely *lambda lifting* and *compilation of pattern-matching*.

Acknowledgements

I am grateful to my supervisor Dr. Robin Cockett for his support over the years. Needless to say, this thesis would not be possible without him. I would like to thank Jonathan Gallagher for his technical insights to the project and being a mentor to me. I would also like to thank my group members and friends Dr. Cristine Bauer, Jonathan Gallagher, Mathhew Burke, Chad Nester, Ben McAdam, JS Lemay, Cole Comfort, and Daniel Satanove for the wonderful conversations, both technical and non technical, we have had over many a lunches and dinners. I am also thankful to them for sharing their valuable suggestions on the various chapters of my thesis. I would like to thank Dr. Robin Cockett, Dr. Richard Zach, and Dr. Mea Wang for serving on my thesis committee.

I would like to thank my wife Anju for her patience and moral support all these years, and for proofreading my thesis. I am indebted to my parents whose indefatigable enthusiasm for my education has made this moment possible.

Dedicated to my loving parents and my wonderful wife Anju

Table of Contents

List of Tables

Chapter 1

Introduction

MPL is a programming language being developed at the University of Calgary. It is a statically typed concurrent programming language which uses *message passing* as the concurrency primitive as opposed to threads, or shared memory. In MPL, concurrency is achieved using concurrency constructs built into the language. This is different from the traditional approach which adds concurrency to a language either by adding operating system primitives to the sequential core of the language or by overloading the basic sequential constructs of the language (e.g the use of monads to perform IO in Haskell).

MPL is based on Cockett and Pastro's ideas [1] which provide a formal theory of message passing as well as a categorical semantics for it. Cockett and Pastro model message passing using a two tier logic: the first tier is a *logic for messages* whose proofs can be thought of as ordinary sequential programs and the second tier is a *logic for message passing* which is built on top of the logic for messages. The logic of messages deals with what we classically view as computation while the logic of message passing deals with the exchange of messages between the communicating processes.

Concurrent programs in MPL consist of processes, which are connected via channels. Interprocess communication takes place by passing messages along channels. A process can perform operations on a channel: for example, a process can send or receive messages on a

channel. The sequence of operations that are permitted on a channel is determined by its *channel type*. If a process tries to perform an operation on a channel which is not permitted by its channel type then a type error is reported during compile time.

The goal of the MPL project was to produce a fully featured, type safe concurrent programming language with intuitive syntax based on Cockett and Pastro's work on the logic of message passing. Type safety for the concurrent programs written in MPL is obtained using the channel types. An MPL program that passes type checking is generally a well behaved program meaning that it will not have a run time error and, when it produces output, will do so with the intended type.

1.1 Structure of an MPL Program

An MPL program consists of sequence of definitions for sequential programs. These definitions are: data and codata definitions, protocol and coprotocol definitions, function definitions, and process definitions.

A programmer can define custom data/codata types for sequential programs. A codata type is a disciplined way to model infinite sequential structures in MPL. A function is used to organise code in the sequential world. A (co)protocol is the analogue of (co)data type in the concurrent world: a programmer can define protocols and coprotocols for concurrent programs. Processes are used to organise code in the concurrent world. They model the exchange of messages on channels. The entry point to an MPL program is the main process, which is a process specified with the `run` command. MPL programs are divided into two levels: sequential MPL program, and concurrent MPL program. An MPL program can have the aforementioned definitions in any order with the following exceptions:

- The last definition in an MPL program is always the *main process* definition defined with the `run` keyword.

- If an MPL definition uses another definition then the used definition should be defined

before the definition that is using it. Exception to this are the cases of *mutually recursive definitions* which are defined using special syntax and the *where* clause.

There are two ways to define a block of code in a programming language. The first approach is to enclose a block of code explicitly using *curly braces* and separate the elements of a block with a *semicolon*. This style of block definition is used in languages like *C* and *Java*. Another approach to define a block is to indent its elements by the same amount. This style of organising the block known as the *offside rule* and is used in languages like *Haskell* and *Python*. The offside rule obviates the need for extra syntax in the form of curly braces and semicolon. For the aforementioned reason, we have gone for offside rule to define a block of code in an MPL program.

MPL allows for proper documentation of programs using single line, and multi line comments. The single line comments in MPL start with the symbol `"--"`. The multi line comments in MPL are enclosed between `"{-"` and `"-}"` symbols.

1.2 An Example of an MPL Program

An example of an MPL program is shown below. The program takes a message on a terminal, and prints the message on another terminal. The program consists of the main process written with the keyword `run`. It has a built in character terminal `charTerm1`. One can interact with an MPL program, write input to or read output from, using these built in channels which are also called *services* in MPL's parlance.

```
run =  => charTerm1
  plug do hput PutString on ch
          put "Hello World!" on ch
          hput CloseS on ch
          halt ch
       putString (| ch => charTerm1)
```

In the main process, two processes are plugged together along a channel `ch`: the first process consists of a block of code consisting of four MPL commands under the `do` keyword and the second process is `putString`. `ch` acts as an input channel for the first process and as an output channel for the second process.

The first process puts a handle `PutString` on the channel `ch`. The handle informs the channel `ch` to expect a string input. Once the handle is put, the string `"Hello World!"` is put on the channel. Once the task of sending the string on the channel is accomplished, the first process is halted. Halting a process requires all the channels asscoiated with the process to be closed. Channels `ch` is closed after putting appropriate handles on it. The handles put on the channel `ch` of the first process are part of MPL's prelude and thus can be used directly.

The second process `putString` is a part of MPL's prelude. `putString` receives a string on its input channel `ch` and prints the string on its output channel `charTerm1` thus printing `"Hello World!"`.

1.3 Stages of Interpretation of an MPL Program

The stages of interpretation of an MPL program are: lexing, parsing, type inference/type checking, pattern-matching compilation, lambda lifting, and the translation of programs to AMPL code which is finally run on the abstract machine for MPL. These stages are shown in Figure 1.1.

The lexing, the parsing, and the type inferencing stages form the front end of the MPL's compiler. The MPL's lexer takes an MPL program, in string representation, as input and produces a list of valid MPL tokens for the program as output. An invalid symbol present in the program results in a lexer error. Once the lexing step is successfully completed, the list of tokens generated by the lexer are passed as input to the MPL's parser which verifies

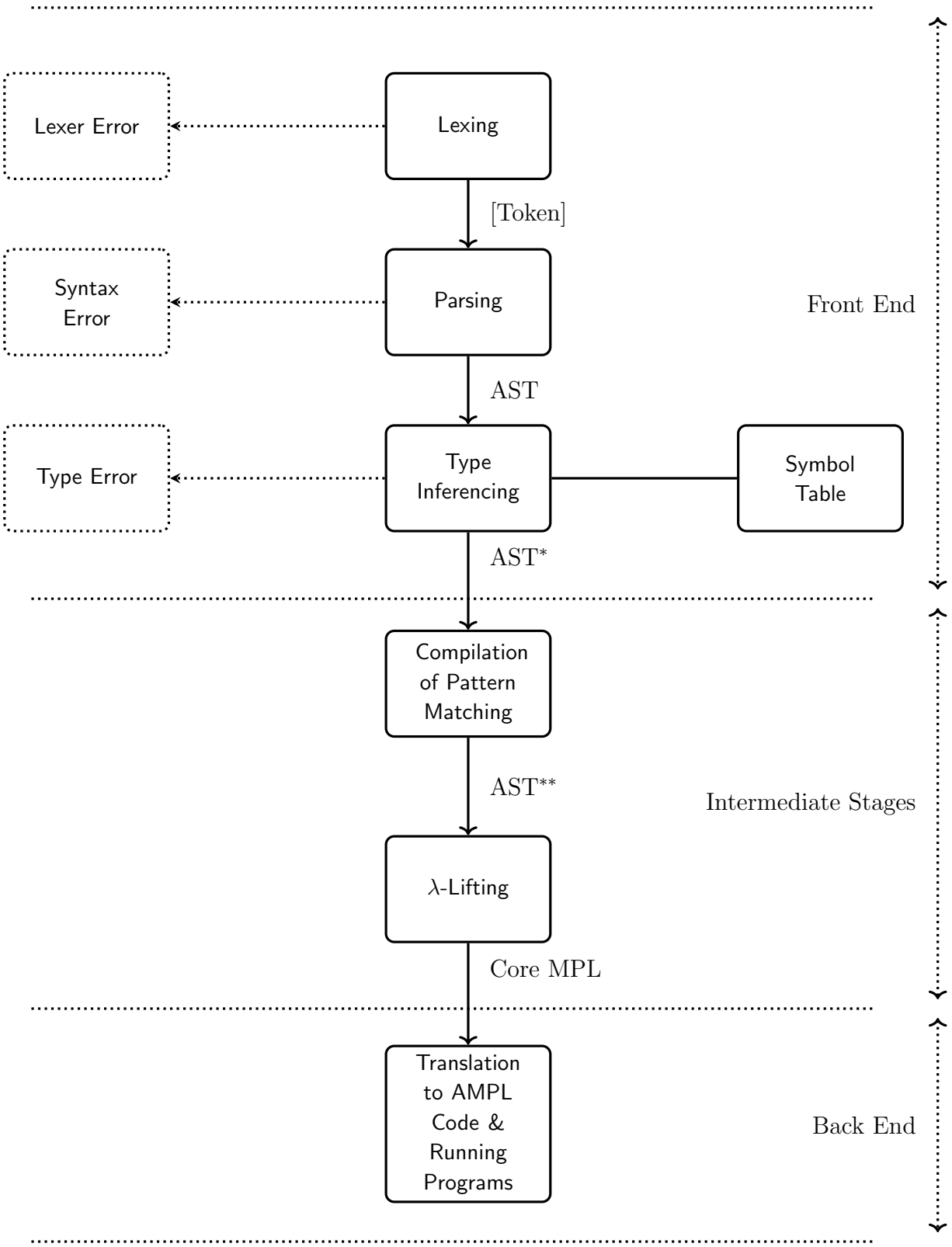

Figure 1.1: Interpretation Stages of MPL

the syntactic correctness of the program. If the program is syntactically correct, the list of tokens are converted to a data structure called the *Abstract Syntax Tree (AST)*. AST is easier to transform than the original string representation of the program and is better suited for the subsequent stages of the compiler.

The AST is the input to the type inference stage, which uses it to verify whether the program is well typed. If the program is well typed then the interpretation of the MPL program is continued otherwise it is halted throwing a type error. The type inferencer ensures that a program with type errors, which constitute a large class of programming errors, are apprehedned in the early stages of interpretation throwing meaningful and precise error messages. The output produced by the type inferencer is represented as AST*, which is a subset of the AST without the type annotations of the functions and the processes.

The *compilation of pattern-matching* and the λ-lifting transformation are the intermediate stages of the MPL's compiler. The AST* is fed as input to the pattern-matching compiler. This stage converts the pattern-matching syntax present in the MPL program to explicit case statements. The output of this stage is represented as AST** which is a subset AST* without the pattern-matching syntax. The AST** is passed as input to the λ-lifting program. The λ-lifting transformation lifts all the local functions to the global scope, which is the outermost scope of an MPL progra.m The output of this stage is *Core MPL*, which is a data structure used for the intermediate representation of MPL programs. Core MPL is a stripped down version of MPL with no pattern matching or where clauses.

The *abstract machine for MPL* (*AMPL*) forms the back end of MPL. AMPL runs MPL code. In order to run an MPL program on AMPL, it must be specified in terms of the instruction set of the abstract machine, i.e in AMPL code.

1.4 Contributions of this Thesis

The main goal behind the MPL project was to produce an implementation of the language. This goal was achieved successfully and the various aspects pertaining to the implementation of MPL form the core of the thesis. Following are the major contributions of the thesis to the MPL project:

- **Lexer and Parser**- A lexer and a parser was implemented for MPL and the syntax of the language was solidified.
- **Type System**- A specification of MPL's type system was provided (in Chapters 4,5, and 6) and implemented.
- **Abstract Machine**- An abstract machine, which runs MPL programs, was designed (in Chapter 9) and implemented.
- **Intermediate Languages**- The compilation of MPL programs to the abstract machine code was achieved using a sequence of intermediate languages namely Core MPL and AMPL Code, described in Appendix E and Appendix F respectively. This results in a modular implementation of the language where any part of the MPL's compiler could easily be exchanged for a more efficient version when such a version becomes available. These intermediate languages are developed as languages in their own right by providing a lexer and a parser for each of them. This technique has two benefits: one can directly write programs in any of the intermediate languages, and one can inspect the series of transformations that a program undergoes in each step of compilation.

At the beginning of each chapter, the specific contributions of that chapter are discussed further.

1.5 Structure of the Thesis

This thesis is divided into two parts: chapters 2, and 3 discuss the MPL language from the user's viewpoint, while chapters 4-9 deal with the implementation details of the language. Specifically:

- **Chapter 2 - Sequential MPL** This chapter deals with the sequential aspects of MPL programs like functions, data types, codata types, pattern-matching, and sequential MPL commands.
- **Chapter 3 - Concurrent MPL** This chapter deals with the concurrent aspects of MPL programs like processes, channels, protocol definitions, coprotocol definitions, and concurrent MPL commands.
- **Chapters 4 to 6 - Type Inferencing** These chapters deal with the type inference process of MPL programs.
- **Chapter 7 - Compilation of Pattern-Matching** This chapter describes the compilation of the pattern-matching syntax to explicit case statements.
- **Chapter 8 - Lambda Lifting Transformation** This chapter describes the λ-lifting transformation which lifts the local MPL functions to the global scope.
- **Chapter 9 - Abstract Machine for MPL** This chapter provides the specification and description of the abstract machine on which MPL programs are run.

1.6 Related Work

In this section, we look at the π-calculus and "session types". π-calculus inspired MPL's choice of language primitives for concurrency and is an important theoretical tool in understanding the modelling of concurrency. Session types are a family of type systems which

provide type safety to concurrent interactions. MPL also has a novel concurrent type system that tries to enforce type discipline in a concurrent program. Thus, an introduction to session types will lay the ground work for the discussion of MPL and typing concurrent programs.

1.6.1 π-Calculus

The π-calculus was introduced by Milner et al. [19, 20]. It is a mathematical model of processes whose interconnections change as they interact. It can be used to model a network of interconnected processes that exchange messsages and where messages can contain links to the active processes.

Syntax of the π-Calculus

Let X be a set of objects called names. The syntax of a variant of π-calculus [19] is presented below (where c and v are any names from X, and P and Q are processes):

P, Q	$::=$	$c(v).P$	Receive v on c, and then run P
		$\bar{c}\langle v\rangle$	Output v on channel c, and then run P
		$P \mid\mid Q$	Parallel composition of P and Q
		$(\nu c)P$	Name Restriction
		$!P$	Repeatedly spawn copies of P
		0	Terminate the process

Reduction Rules

$P \to Q$ means that P can be transformed to Q in one step of computation. The reduction rules are:

- **Communication Rule** - Suppose there are two parallel processes P, and Q. If value v is output on the channel c in the process P and the process Q is expecting a value x on the channel c, then both the input and the output operations are executed in one step, and in the process Q the parameter x is replaced with the value v.

$$\bar{c}\langle v\rangle.P \parallel c(x).Q \;\rightarrow\; P \parallel Q\{v/x\}$$

- **Parallel Rule** - If there are two processes running in parallel, one can do a one-step reduction of one of the parallel processes without affecting the other.

$$\frac{P \;\rightarrow\; Q}{P \parallel P' \;\rightarrow\; Q \parallel P'}$$

- **Restriction Rule** - This rule ensures that reduction can proceed underneath a restriction. Creation of a new name in a process is also called restriction.

$$\frac{P \;\rightarrow\; Q}{(\nu x)P \;\rightarrow\; (\nu x)Q}$$

- **Structural Congruence Rule** - Structurally congruent processes have the same reductions.

$$\frac{P \equiv P' \qquad P' \;\rightarrow\; Q' \qquad Q' \equiv Q}{P \;\rightarrow\; Q}$$

Programming Languages Based on the π-Calculus

Some examples of concurrent programming languages based on π-calculus are:

- **Pict** - Pict [37] is a concurrent programming language developed by Pierce and Turner. It is based on the π-calculus. It is a language in the ML tradition, formed by adding a layer of convenient syntactic sugar and a static type system to a tiny core. The core

language - an asynchronous variant of Milner, Parrow, and Walker's pi-calculus - has been used as a theoretical foundation for a broad class of concurrent computations. The core language contains two kinds of entities: processes (agents) and channels (names), where processes use channels to communicate with other processes.The type system of Pict integrates features like higher-order polymorphism, simple recursive types, and subtyping.

- **TyCO** - TyCO (Typed Concurrenct Object) is a programming language developed by Vasconcelos [38]. It is an implicitly typed polymorphic concurrent language based on an extension of the asynchronous pi-calculus featuring first class objects, asynchronous messages and process definitions. It provides a model for a concurrent object-based language, which combines the benefits of the formal framework of process calculi with the characteristics of Hewitt's actor system [39].

1.6.2 Session Types

Session types were introduced by Honda et al. [22, 23]. Session types are a family of type systems, which bring type safety to concurrent interactions, which may be concurrent programs modelled using a process algebra like π-calculus or concurrent programs written in a programming language like Haskell. A *protocol* represents an agreement on how participating systems interact with each other. In this regard, they are similar to MPL's *channel types* and protocols.

In Honda et al. [22, 23], an extension of the π-calculus with session types is proposed. Later, Gay and Hole [24] introduced a notion of subtyping for session types for a more conventional π-calculus. Session types were originally described in a two processes setting. n-ary session types were introduced in [25]. Honda et al. [28, 29] presented the Scribble framework in which one could specify protocols in concurrent programs and validate the programs against the type checker. Scribble supports binding for various high level languages such as ML, Java, Python, and C++.

Baltazar et al. combine session types with linear refinements resulting in an original system of linearly refined session types [26]. Refinement types are a form of dependent types which allow attaching formulae to types [27], thus specifying properties of values in programs. The refinement type $\{x : T \mid A\}$ represents a value v of type T that must respect formula A. Formula A may refer to v via variable x. For example, the type $\{x : Integer \mid x \geq 0\}$ describes a natural number.

Session Types in Functional Programming Languages

Some implementations of session types in functional programming languages are:

- **Haskell** - Neubauer and Thiemann [30] presented an implementation of session types in Haskell. In this impementation the session types are encoded in terms of type classes with functional dependencies using the *session monad*. In a similar work, Sackman and Eisenbach [31] describe the encoding of session types in Haskell using a Domain Specific Language (DSL), which works within an extended *Monad* type class. The use of DSL allows to assign labels to session types, or fragments of session types, and to refer them using these labels.

- **Verifying Cryptographic Protocols with Session Types** - Bhargavan et al. describe the design and implementation of a compiler in ML that, given high-level multi-party session descriptions, generates custom cryptographic protocols [32]. The compiler generates code for sending and receiving these messages, with cryptographic operations and checks, in order to enforce these guarantees against any adversary that may control both the network and some session participants. The code generated by the compiler is secured by relying on the session types. Most of the proof is performed by mechanized type checking.

Session Types in Object-Oriented Programming Languages

Some implementations of session types in object oriented languages are:

- **Sing#** - It is a type-safe, object-oriented, and garbage collected programming language [33], which is a variant of the programming language C#. Sing# uses message passing via shared-memory as the communication mechanism between processes. This language was used to write the Singularity operating system [34] , ensuring process isolation (a process cannot access or corrupt data or code of another process) and inter-process communication (processes may exchange messages and signal events), two important services of the operating system. In Sing#, data is exchanged over bidirectional channels where each channel consists of exactly two endpoints (called Imp and Exp). Channel endpoints can be sent in messages over channels meaning that communication network can evolve dynamically. Channel communication is governed by statically verified channel contracts that describe messages, message argument types, and valid message interaction sequences as finite state machines similar to session types.

- **SJ** - Hu et al. [35] describe a session based extension of Java, called SJ. They present the implementation and runtime for a session-based distributed programming featuring asynchronous message passing, delegation, and session subtyping and interleaving, combined with class downloading and failure handling. The compilation-runtime framework of the language maps session abstraction onto underlying transports and guarantees communication safety through static and dynamic session type checking.

- **Mool** - Mool is a small object-oriented language similar to Java, with support for concurrency. It was developed by Campos and Vasconcelos [40]. Mool formalizes protocols, called usage types, that define how and when the method of a class should be called.

Session Types in Imperative Languages

Ng et al. [36] describe a programming tool chain for message-passing parallel algorithms, which ensures for any typable programs, deadlock-freedom, communication safety, and global

progress through a static checking. The methodology is embodied as a multiparty session-based programming environment for C and its runtime libraries, which is called Session C. The Session C framework uses the programming language Scribble to describe communication protocols in the form of multiparty session types.

Comparing MPL with Session Types

As seen in Section 1.6.2, *session types* are not a specific type system rather they are a family of type systems for modelling concurrent communications between processes. One can thus choose a programming language or a process algebra like Haskell, Java, or π-calculus, and then add a session type to them in order to type check the concurrent communications in the system. MPL has typed communication between the processes and thus in principle its type system is an example of a session type. In this section we compare MPL as a programming language with its type system to the programming languages with session types as a type system for concurrent communication, the ease of development of concurrent programs being the evaluation criterion.

In Section 1, we noted that adding operating system primitives to the sequential core of the language or overloading the sequential constructs of the language are traditionally the mechanisms of achieving concurrency in an already existing programming language. Most of the programming languages with session types achieve concurrency using these two mechanisms. We are of the opinion that this might not be the best approach, often vindicated by the fact that concurrent programs are difficult to design, implement, and most importantly get right. Even the added assurances of the session types are not enough to tackle these inherent difficulties. MPL was designed with concurrency in mind which manifests itself through the selection of message passing as the concurrency primitive, and having abstract language primitives suitable for concurrent program development. These design choices result in a natural modelling of concurrent configurations easing the development of concurrent programs.

MPL is a full-fledged language, and, unlike many implementations of session types, not an optional extension to a language. This results in a cleaner syntax for the concurrent aspects of the program. MPL also has a sequential part, which is a sophisticated strongly typed functional programming language. The sequential computations are separated from concurrent computations. At the same time, it is easy to call these sequential computations in the form of functions inside concurrent processes. Thus, MPL provides the support of a functional programming language, assists in modular development of the concurrent and sequential aspects of a program, and provides for a convenient interplay of the two aforementioned aspects of the program.

Thus, MPL provides other benfits as well to the programmers in addition to the type safety guarantees of the session types.

Chapter 2

Sequential MPL

In the first chapter we saw that MPL programs are divided into two levels: concurrent MPL programs and sequential MPL programs. In this chapter, the various aspects of sequential MPL programs are discussed.

Sequential MPL programs consist of functions which use data type and codata type definitions. Function definitions use pattern-matching and the various sequential MPL constructs in their body. Sequential MPL constructs, therefore, can be divided into three parts: constructs that deal with data types, constructs that deal with codata types, and constructs which are general to all sequential programs. Data types and related constructs are described in Section 2.1. Codata types and related constructs are described in Section 2.2. Section 2.3 describes the remaining sequential MPL constructs. Section 2.4 introduces the pattern-matching syntax of MPL.

The syntax for sequential MPL follows the syntax of functional languages fairly closely. However, it also uses conductive data types, and thus is influenced by the syntax for *Charity* [4]. Charity doesn't have mutually recursive data types which MPL does, both on the inductive and coinductive side. This meant that the syntax for inductive and coinductive data types and their associated constructs had to be designed.

2.1 Data Types and Constructs for Data Types

MPL allows the programmers to define data types. The syntax of a data definition is:

$$\begin{array}{lllll} \texttt{data} \quad D(A_1,\ldots,A_k) \ \rightarrow \ Z = & C_1 \ :: \ T_{1,1},\ldots,T_{1,n_1} & \rightarrow & Z \\ & \vdots \qquad\qquad \vdots \qquad\qquad \vdots & & \\ & C_m \ :: \ T_{m,1},\ldots,T_{m,n_m} & \rightarrow & Z \end{array}$$

The type constructor of the data type is D and $C_1,\ldots,C_m$ are the constructors of the data type. The data type $D(A_1,\ldots,A_k)$ is polymorphic in type variables $\{A_1,\ldots,A_k\}$. The type variable Z is the *state variable*. The union of all the type variables used in the different constructors of the data type is $\{Z, A_1,\ldots,A_k\}$. Thus, for any valid type expression $T_{p,q}$ used in the data definition:

$$\mathsf{TypeVar}\ (T_{p,q}) \quad \subseteq \quad \{Z, A_1,\ldots,A_k\}.$$

The type of the phrase of a fold function corresponding to the constructor C_i is directly given by the type expression corresponding to that constructor in the data definition:

$$T_{i,1},\ldots,T_{i,n_i} \quad \rightarrow \quad Z.$$

The type of a constructor C_i of the data type D is given by substituting the state variable by the data type name inside the type expression corresponding to that constructor in the data definition:

$$(T_{i,1},\ldots,T_{i,n_i} \quad \rightarrow \quad Z)\ [D(A_1,\ldots,A_k)/Z].$$

2.1.1 Examples of Data Types

In this section, we look at syntax of data type definitions through some examples. The data definitions can use the base types likes `Char` (characters), `Int` (integers), and `Float` (floating point numbers).

Boolean Data Type

The Boolean data type is defined in MPL as:

```
data Bool -> C = True,False  :: -> C
```

where `True` and `False` are the two constructors of the data type `Bool`. `Bool` doesn't have any type variables. The types of the constructors `True` and `False` are obtained by replacing the state variable `C` with `Bool` so that:

```
True  :: -> Bool
False :: -> Bool
```

Natural Number Data Type

The natural number data type is defined in MPL as:

```
data Nat -> C = Zero  ::   -> C
              Succ  :: C -> C
```

The `Nat` data type is made up of two constructors, `Zero` and `Succ`. The types of the constructors are:

```
Zero  ::     -> Nat
Succ  :: Nat -> Nat
```

Natural numbers can be generated using the constructors `Zero` and `Succ` of the `Nat` data type:

```
Zero, Succ(Zero), Succ(Succ(Zero)),...
```

where `Zero` represents 0, `Succ(Zero)` represents 1, `Succ(Succ(Zero))` represents 2 and so on.

List Data Type

The `List` data type is defined as:

```
data List(A) -> C = Nil   ::     -> C
                    Cons  :: A,C -> C
```

The `List` data type is polymorphic in the type variable `A` meaning that it can be used to store elements of any type `A` in a list. `List` has two constructors, `Nil` and `Cons` whose types are:

```
Nil   ::  -> List(A)
Cons  :: A,List(A) -> List(A)
```

Since lists are very commonly used in MPL, a convenient syntax is provided for its two constructors: `[]` for the `Nil` constructor and `:` for the `Cons` constructor. Some examples of lists using the constructors `Nil` and `Cons` are:

```
Nil,Cons(1,Nil),Cons(1,Cons(2,Nil)),Cons(1,Cons(2,Cons(3,Nil))),...
```

The above list can equivalently be written in the convenient syntax of `List` as:

```
[], 1:[], 1:2:[], 1:2:3:[],...
```

where `Nil` or `[]` represents an empty list, `Cons(1,Nil)` or `1:[]` represents a singleton list containing integer 1 etc.

Either Data Type

The `Either` data type is defined as:

```
data Either(A,B) -> C = Left   :: A -> C
                       Right  :: B -> C
```

where `Left` and `Right` are the two constructors of the data type. The types of the constructors are:

```
Left   :: A -> Either(A,B)
Right  :: B -> Either(A,B)
```

The `Either` data type can be used as the output type of a function that can return two different types as output, say `A` and `B`. When the function returns the value of type `A`, the `Left` constructor is used and when the function returns the value of type `B`, the `Right` constructor is used. This data type is often used as the output type of a function that can fail for some inputs. The function returns an error message in the `Left` constructor and the normal output in the `Right` constructor.

2.1.2 Mutually Recursive Data Types

In addition to normal data types, MPL allows the programmers to define mutually recursive data types using the following syntax:

```
data
```

$$
\begin{aligned}
&D_1(A_{11}, \ldots, A_{1k}) \;\rightarrow\; Z_1 = \\
&\quad C_{11} \;::\; T_{11,1}, \ldots, T_{11,n_{11}} \quad \rightarrow \quad Z_1 \\
&\quad \vdots \qquad\quad \vdots \qquad\quad \vdots \\
&\quad C_{1m} \;::\; T_{1m,1}, \ldots, T_{1m,n_{1m}} \quad \rightarrow \quad Z_1
\end{aligned}
$$

```
and
```

$$
\begin{array}{l}
\vdots \\
\texttt{and} \\
D_q(A_{q1}, \ldots, A_{ql}) \ \rightarrow \ Z_q = \\
\qquad \begin{array}{lcll}
C_{q1} & :: & T_{q1,1}, \ldots, T_{q1,n_{q1}} & \rightarrow \quad Z_q \\
\vdots & & \vdots & \vdots \\
C_{qr} & :: & T_{qr,1}, \ldots, T_{qr,n_{qr}} & \rightarrow \quad Z_q
\end{array}
\end{array}
$$

where data definitions $D_1, \ldots, D_q$, separated by the keyword `and`, are mutually recursive with each other meaning that a *call graph* of the state variables of the corresponding data types will be connected. Corresponding to a mutually recursive data definition there is a call graph, which indicates which state variable are used by other state variables. It is defined as:

- **Nodes**: State variables of the data types
- **Edges**: There is a from node Z_i to node Z_j when there is a constructor C_{ir} such that $Z_j \in T_{ir,k}$ for some r and k.

Casing on a mutually recursive data type means casing on just the constructors of one of the types. However, folding on a mutually recursive data type means that all the types which are reachable from that data type in the call graph need to have phrases corresponding to their constructors. In the data definition above, the data types $D_2, \ldots, D_q$ should be reachable from the first data type D_1 otherwise a warning is issued saying that some data is not *reachable*.

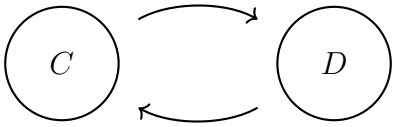

Figure 2.1: Call Graph of the Data Types of Tree and Forest and of Zig and Zag

2.1.3 Examples of Mutually Recursive Data Types

Tree and Forest

Mutually recursive data types `Tree` and `Forest` [2] are defined as:

```
datatype
    Tree(A) -> C   = Empty::      -> C
                   Node :: A,D -> C
    and
    Forest(A) -> D = Nil  ::      -> D
                   Cons :: C,D -> D
```

A tree which can have mutiple number of children is called a *variadic tree* [2]. The mutually recursive data definitions above defines a variadic tree to be either empty, or a node collecting together a forest to form a tree, and a forest to be either empty or a variadic tree together with another forest. A call graph of the state variables of the data types `Tree` and `Forest` is shown in Figure 2.1. Note that the nodes `C` and `D` of the call graph are connected so that the data types `Tree` and `Forest` are mutually recursive. The types of the constructors are obtained by replacing the state variables `C` and `D` with `Tree(A)` and `Forest(A)` respectively in the type expressions corresponding to the constructors:

```
Empty ::      -> Tree(A)
Node  :: A,Forest(A) -> Tree(A)
Nil   ::      -> Forest (A)
Cons  :: Tree(A),Forest(A) -> Forest(A)
```

Mutually Recursive Data Types Zig and Zag

Mutually recursive data types `Zig` and `Zag` [3] can be used to define a list with elements of the alternating types. They are defined as:

```
datatype
    Zig(A,B) -> C = Nil  ::     -> C
                  Cins :: A,D -> C
    and
    Zag(A,B) -> D = Nal  ::     -> D
                  Cans :: B,C -> D
```

The call graph for `Zig` and `Zag` data types will have the same structure, shown in Figure 2.1, as the call graph of `Tree` and `Forest` data types. The types of the constructors are obtained by replacing the state variables `C` and `D` with `Zig(A,B)` and `Zag(A,B)` respectively in the type expressions corresponding to the constructors:

```
Nil   ::     -> Zig(A,B)
Cins  :: A,Zag(A,B) -> Zig(A,B)
Nal   ::     -> Zag(A,B)
Cans  :: B,Zig(A,B) -> Zag(A,B)
```

2.1.4 Constructs for Data Types

The basic sequential MPL programming constructs for the data types are: the case, and the fold. These constructs are described in this section.

The case Construct

The case construct is the basic MPL construct used with data types. The syntax of a case construct for a data type D defined in Section 2.1 is:

$$
\begin{array}{l}
\mathsf{case}\ \ t\ \ \mathsf{of} \\
\qquad C_1\ \ (v_{1,1}, \ldots, v_{1,n_1}) \ \ \rightarrow t_1 \\
\qquad\qquad \vdots \\
\qquad C_m\ \ (v_{m,1}, \ldots, v_{m,n_m}) \ \ \rightarrow t_m
\end{array}
$$

The type of the sequential term t is the data type D. $C_1, \ldots, C_m$ are the constructors of a data type D. All the constructors of the data type D must be present in the case construct (or the case construct must have a `default` term). The case construct branches on the constructors of the data type D and based on what t evaluates to, it selects a branch. Once a particular branch, say C_k, is selected, the corresponding term t_k is executed.

Three examples that use the case construct, namely on the data types `Nat`, `List`, and the mutually recursive data types `Forest` and `Tree`, are discussed next. In the first example below, `nat2Int` function is defined which translates the unary representation of a number into its integer representation:

```
-- convert Nat to Integer
nat2Int :: Nat -> Int =
  nat -> case nat of
           Zero    -> 0
           Succ(n) -> 1 + nat2Int(n)
```

`nat2Int` is a recursive function which returns the integer zero for the `Zero` constructor and for the `Succ(n)` branch, the function adds one for every `Succ` constructor.

The function append takes two lists as inputs and produces the appended list as output, is defined as:

```
-- append two lists
append :: List(A),List(A) -> List(A) =
  t1,t2 -> case t1 of
             []    -> t2
             x:xs  -> x:(append(xs,t2))
```

The function sizeTree cases on the data type Tree(A). Casing on a mutually recursive data type results in mutually recursive functions as shown below:

```
defn
    -- size of a tree
    sizeTree :: Tree(A) -> Int =
      t -> case t of
              Empty -> 0
              Node(t',f) -> 1 + sizeForest(f)


    -- size of a forest
    sizeForest :: Forest(A) -> Int =
      f -> case f of
              Nil -> 0
              Cons(t,f') -> size_tree(t) + size_forest(f')
```

The defn construct of MPL allows the definition of mutually recursive functions besides being used to provide a module system in MPL.

The fold Construct

The fold construct is a sequential MPL construct that allows programmers to use recursion in a disciplined manner. Functions written using fold construct belong to a set of functions called higher-order primitive recursive functions which always terminate (provided their phrases do). All the functions written using folds can also be implemented using normal recursion.

The syntax of the fold construct for the data type D defined in Section 2.1 is:

$$
\begin{array}{l}
\mathsf{fold}\ \ t\ \ \mathsf{of} \\
\qquad C_1\ \ :\ \ v_{1,1}, \ldots, v_{1,n_1} \quad \rightarrow t_1 \\
\qquad\qquad \vdots \\
\qquad C_m\ \ :\ \ v_{m,1}, \ldots, v_{1,n_m} \quad \rightarrow t_m
\end{array}
$$

The term t is being folded over: t evaluates to an element of the data type D. $C_1, \ldots, C_m$ are the constructors associated with the data type D. All the constructors of D must be present in a fold construct. For constructor C_i, $v_{i,1}, \ldots, v_{i,n_i}$ is the corresponding sequence of argument bound in t_i giving the i^{th} phrase of the fold. The fold selects the phrase determined by the current constructor and executes the term corresponding to the constructor in the phrase. The types of the phrases of the fold are given by the type annotations for the constructor in the data definition.

We illustrate the fold construct with three functions. Note that the same functions were written using the case construct in Section 2.1.4.

The first example is `convNat2Int` function. The fold types for the constructors `Zero` and `Succ` when a term of type `Nat` is folded are:

```
Zero ::   -> C
Succ :: C -> C
```

These types are directly obtained from the MPL data definition of the `Nat` data type.

The `convNat2Int` function written using the fold construct is:

```
-- convert Nat representation to Integer
nat2Int:: Nat -> Int =
  nat -> fold nat of
           Zero :   -> 0
           Succ : n -> 1 + n
```

where, for the `Zero` constructor, the integer zero is returned and, for the `Succ` constructor, its argument incremented by one.

The append written using the fold is:

```
-- append two lists
append :: List(A),List(A) -> List(A) =
  t1,t2 -> fold t1 of
             Nil  :    -> t2
             Cons :x,r -> x:r
```

The `sizeTree` function written using the fold construct is:

```
-- size of tree
sizeTree :: Tree(A) -> Int =
  t -> fold t of
           Empty :     -> 0
           Node  :n,f  -> 1 + f
           Nil   :     -> 0
           Cons  :t',f -> t' + f
```

In the above example, the data type `Tree` is the subject of the fold. Therefore, all the constructors for `Tree`, as well as the data type `Forest` - which `Tree` is mutually recursive with- must have phrases in the fold.

2.2 Codata Type and Constructs for Codata Type

MPL allows the programmers to define codata types: this is a facility lacking in most functional languages. A language which has codata types is Charity [4]. Codata types provide the means to represent potentially infinite structures that are evaluated lazily.

Consider a codata type $C(A_1, \ldots, A_k)$ having destructors $D_1, \ldots, D_m$ defined below:

$$\begin{array}{lllll} \mathsf{codata}\ Z \ \rightarrow\ C(A_1,\ldots,A_k) = & D_1 \ :: \ Z,\ T_{1,1},\ldots, & T_{1,n_1} & \rightarrow & P_1 \\ & \vdots & \vdots & \vdots & \\ & D_m \ :: \ Z,\ T_{m,1},\ldots,T_{1,n_m} & & \rightarrow & P_m \end{array}$$

The name of the codata type is C and $D_1, \ldots, D_m$ are the m destructors of the codata type C. The codata type $C(A_1, \ldots, A_k)$ is polymorphic in type variables $\{A_1, \ldots, A_k\}$. Z is the state variable. The union of all the type variables used in the different destructors of the codata type is $\{Z, A_1, \ldots, A_k\}$

Consider destructor D_i: its first argument must be the state variable and the next n_i arguments, whose types are represented by $T_{i,1}, \ldots, T_{i,n_i}$, represent the higher-order arguments. If there are no higher-order arguments, then C is a *first-order* data type. Each $T_{i,j}$, for $1 \leq j \leq n_i$, is a type expression such that:

$$\texttt{TypeVar}\ (T_{i,j}) \quad \subseteq \quad \{A_1, \ldots, A_k\}$$

The type expression P_i is the output type of destructor D_i and is:

$$\texttt{TypeVar}\ (P_i) \quad \subseteq \quad \{Z, A_1, \ldots, A_k\}$$

For the destructor D_i, the type of the phrase in the unfold function definition of the codata type C is:

$$T_{i,1}, \ldots, T_{i,n_i} \ \rightarrow \ P_i$$

For the destructor D_i of the codata type C, the type of the destructor is given by substituting the state variable Z with $C(A_1, \ldots, A_k)$ in the body of the type expression corresponding to that destructor in the codata definition.

The record of a codata type contains all the destructors of the codata type. The unfold of a codata type contains phrases corresponding to all the destructors of the codata type.

2.2.1 Examples of Codata Types

InfList Codata Type

`InfList` codata type is defined as:

```
codata C -> InfList(A) = Head :: C -> A
                         Tail :: C -> C
```

As the name suggests `InfList(A)` codata type can be used to store an infinite list of things of type `A`. `InfList(A)` has two destructors, `Head` and `Tail`. `Head` and `Tail` are functions, the types of which can be obtained by substituting `C` with `InfList(A)` in the corresponding type expressions of the destructors. The types of the `Head` and `Tail` destructors are:

```
Head :: InfList(A) -> A
Tail :: InfList(A) -> InfList(A)
```

Lazy Triple Codata Type

Triples can be defined in MPL using codata types as:

```
codata C -> Triple(D,E,F) = P0 :: C -> D
                          P1 :: C -> E
                          P2 :: C -> F
```

`Triple` has three destructors, `P0`, `P1`, `P2` which act as the three projections of the triple. Thus, these destructors can respectively be used to get the first, second and the third elements of the triple. The types of the three destructors are:

```
P0 :: Triple(D,E,F) -> D
P1 :: Triple(D,E,F) -> E
P2 :: Triple(D,E,F) -> F
```

By using codata types, tuples of any length can be used in an MPL program. Since tuples/products are useful programming constructs to have in a language, MPL allows the programmers to use a n-tuple without defining the corresponding codata type for it. `#i` can used to project the i^{th} element (for $1 \leq i \leq n$) from the n-tuple. For example, a pair of a integer and a float can be represented as `(1,2.0)` using the built-in syntax for tuples. `1` can be projected out of the pair using `#i` operator using the syntax: `#1 (1,2.0)`.

Exponential Codata Type

Higher-order functions can be defined in MPL using a simple higher order codata type called `Exp` (exponential) defined as:

```
codata C -> Exp(A,B) = App :: C,A -> B
```

`Exp(A,B)` is the type of total functions from type `A` to type `B` and is also written as `A` $\Rightarrow$ `B` in literature. The type of the destructor `App` is:

```
App :: Exp(A,B),A -> B
```

Section 2.2.3) shows an example using the `Exp` data type to implement a higher order function.

2.2.2 Mutually Recursive Codata Types

As for data types, one can also define mutually recursive codata types. For a list of data types to be mutually recursive, the call graph of the state variables of the codata types should be reachable from the initial state variable.

The record of a codata type C, which is mutually recursive with other codata types, will only contain the destructors of C. However, the unfold for C will contain phrases consisting of the destructors of all the codata types which are mutually recursive with C.

Mutually Recursive InfZig and InfZag Codata Types

The mutually recursive codata types `InfZig(A,B)` and `InfZag(A,B)` can be used to generate an infinite list with elements of alternating types. They are defined as:

```
codata
    C -> InfZig(A,B) = HeadA :: C -> A
                       TailA :: C -> D
    and
    D -> InfZag(A,B) = HeadB :: D -> B
                       TailB :: D -> C
```

2.2.3 Constructs for Codata Types

The basic codata type constructs are: record, destructor and unfold.

The Record and the Destructor Constructs

The record construct is used to hold the value of a codata type. The syntax of a record for the codata type C, defined in 2.2, is:

$$(D_1 := v_{1,1}, \ldots, v_{1,n_1} \rightarrow t_1, \ldots, D_m := v_{m,1}, \ldots, v_{m,n_m} \rightarrow t_m)$$

where D_i is the i^{th} destructor of the codata type. If the destructor D_i doesn't take any argument, then the phrase corresponding to the destructor in record can be represented as:

$$D_i = t_i$$

For example, the following syntax creates a record of the `Triple` codata type:

```
(P0 := 1 , P1 := 2.3 , P2 := 'c')
```

Destructors retrieve value from a codata type. The destructor has the form:

$$D_i\ (r)$$

where r is a record term and $1 \leq i \leq n$. Suppose the record is internally of the form:

$$r := (D_1 := u_1 \rightarrow t_1, \ldots, D_n := u_n \rightarrow t_n)$$

Then, the destructor term $D_i\ (r, u_i)$ evaluates to t_i. Thus, in the below term if `r` is `(P0 := 1 , P1 := 2.3 , P2 := 'c')`, then `P2 (r)` = `'c'`.

The function `firstN` defined below takes a number `n` and generates a list of `n` integers starting from 0. It generates the integer list by creating an infinite list of numbers and then taking `n` integers from that infinite list. Function `firstN` uses two functions `genInfNats` and `getNumList`.

```
-- create a record of infinite list of numbers
genInfNats :: Int -> InfList(Int) =
  n -> (Head := n , Tail := genInfNats(n+1))

-- get specified number of integers from the infinite list
-- and put the integers in a list
getNumList :: Int, InfList(Int) -> [Int] =
  n,infL -> case (n == 0) of
              True  -> []
              False -> Head(infL) : getNumList (n-1,Tail(infL))

-- generate a list of first n numbers starting from 0.
firstN :: Int -> [Int] =
  n -> getNumList(n,genInfNats(0))
```

`getInfNats` creates a record of `InfList` codata type. `getNumList` uses the destructors `Head` and `Tail` to extract a specified number of integers from the record. Finally, function `firstN` calls `getNumList` with the number of integers needed and a record of infinite list of integers starting from 0.

The function `mapSquare` defined below takes a list of integers and returns a list of integers where every element of the list has been squared. The example illustrates the use of the `Exp` codata type in implementing higher-order functions in MPL:

```
-- apply a function from A -> B over every element of a list of
-- type A and give back a list of type B.
mapFun :: [A],Exp(A,B) -> [B] =
```

```
  [],_      -> []
  (x:xs),f -> App(f,x):(mapFun(f,xs,))

-- map the square function over every element of the list
squareFun :: [Int] -> [Int] =
  list -> mapFun (list,(App := n -> n*n))
```

`mapFun` takes in a list and the function to be applied to the every element of the list, which in this case is $\lambda x.x * x$.

The unfold Construct

The codata types are built using the unfold construct. The syntax of an unfold for the codata type C, defined in Section 2.2, is:

$$\begin{array}{l} \mathsf{unfold}\ \ t\ \ \mathsf{of} \\ \quad s\ \ \Rightarrow \\ \qquad D_1\ :\ u_{1,1}, \ldots, u_{1,n_1}\ \ \rightarrow\ \ t_1 \\ \qquad\qquad \vdots \\ \qquad D_m\ :\ u_{m,1}, \ldots, u_{m,n_m}\ \rightarrow\ \ t_m \end{array}$$

t is the initial value of the state which is to be unfolded. $D_1, \ldots, D_m$ are the destructors of the codata type C. $u_{i,1}, \ldots, u_{i,n_i}$ is the sequence of n_i arguments corresponding to the destructor D_i and t_i is the corresponding term which produces the new state. The set of phrases corresponding to the different destructors are also known as the threads of the unfold. s is a state which is accessible to all destructors $D_1, \ldots, D_m$ in the unfold.

The genInfNats function defined below generates a list of infinite numbers starting from num using the unfold construct. Note that genInfNats was implemented in the Section 2.2.3 using records.

```
-- infinite list of numbers starting from 0
genInfNats :: Int -> InfList(Int) =
  num -> unfold of num
           n =>
             Head : n
             Tail : n + 1
```

Function natIntInfL, defined below, is used to create an infinite list of alternating natural numbers and their corresponding integer representation:

Zero,0,Succ(Zero),1,Succ(Succ(Zero)),2,...

natIntInfL illustrates the use of the unfold construct over mutually recursive codata types.

```
-- infinite list of alternating Nats and Integers starting from
-- Zero and 0 respectively
natIntInfL ::  -> InfZig(Nat,Int) =
    -> unfold (Zero,0) of
           (n,i) =>
               HeadA : n
               TailA : (i,Succ(n))
           (i,n) =>
               HeadB : i
               TailB : (n,i+1)
```

Function natIntInfL outputs a record of type InfZig(Nat,Int). The initial values unfolded, which correspond to the codata types InfZig and InfZag, are Zero and 0 respectively. Destructor HeadA of the InfZig codata type stores the Nat part of its corresponding

state in the record. Destructor `TailA` updates the `Nat` part of its corresponding state by a successor, thereby updating the state passed to the `InfZag` codata type. Similarly, destructor `HeadB` of the codata type `InfZag` stores the integer part of its state in the record and the destructor `TailB` increases the integer part of its corresponding state by a value one, thereby updating the state passed to the `InfZig` codata type.

2.3 Other Sequential MPL Constructs

Apart from the MPL constructs used with the data and the codata types, there are some other sequential MPL constructs as well. They are the where clause, the if-then-else construct, the switch construct, function calls, variables, and constants.

2.3.1 If-Then-Else

The `if-then-else` is a sequential construct that takes three arguments. It syntax is:

$$\texttt{if}\ \ b\ \ \texttt{then}\ \ t_1\ \ \texttt{else}\ \ t_2$$

where the first argument `b` is a sequential MPL term that evaluates to a boolean value. If the boolean value evaluates to `true` then t_1 is executed else t_2 is executed.

2.3.2 Where

In MPL, the where clause is used to define local functions attached to a term.

Function `fah2Kel`, defined below, shows an example of an MPL function that uses the where clause. The function takes a temperature in degree Fahrenheit and converts it to the corresponding temperature in Kelvin. This conversion is achieved by first converting degree Fahrenheit to degree Celsius and then converting degree Celsius to Kelvin.

```
-- convert degree Fahrenheit to Kelvin
```

```
fah2Kel :: Double -> Double =
  fah -> kel()
      where
        kel =
              -> fah2Cel(fah) + 273.15
        fah2Cel =
          fah -> (5*fah - 160)/9
```

Function `kel` and `fah2Cel` are local function defined using the where clause, which is attached to the function call. The scope of the functions defined using the where clause is the other functions defined in the where clause and the term to which the where clause is attached.

2.3.3 Switch

The switch term, sometimes known as guards in the vocabulary of functional programming, is of the form:

$$
\begin{array}{l}
\texttt{switch} \\
\quad p_1 \ \rightarrow \ t_1 \\
\qquad \vdots \\
\quad p_n \ \rightarrow \ t_n \\
\quad \texttt{default} \ \rightarrow \ t_{def}
\end{array}
$$

`switch` consists of a list of pairs of terms. Each pair (p_i, t_i) corresponds to a line in the body of the switch construct. All the first elements of the list of pairs must return a boolean value. The first term from $p_1, \ldots, p_n$ to returns a `True` will have its corresponding term executed.

If none of $p_1, \ldots, p_n$ return a true then the term t_{def} corresponding to the `default` case is executed.

Function `dayName`, defined below, shows an example where the `switch` construct is used. It takes an integer representing the number of a day in a week as the input. If the integer is between one and seven, the function returns the corresponding constructor from the the `Day` data type along with the `SS` (succeess) constructor of the `SF` (success or fail) data type. If the day number is greater than seven, the `FF` (fail) constructor is returned. Note that in the definition of the `Day` type, the constructors are grouped together: MPL's syntax allows for the grouping of constructors having the same types.

```
-- Success or Fail Constructor
data SF(A) -> C = SS :: A -> C -- Success constructor
                  FF ::   -> C -- Fail constructor


-- data type representing week days
data Day -> C = Mon , Tue , Wed, Thu , Fri , Sat , Sun :: -> C


-- convert day numbers of a week to their day names
dayName :: Int -> SF(Day) =
  day -> switch
              day == 1 -> SS(Mon)
              day == 2 -> SS(Tue)
              day == 3 -> SS(Wed)
              day == 4 -> SS(Thu)
              day == 5 -> SS(Fri)
              day == 6 -> SS(Sat)
              day == 7 -> SS(Sun)
              default  -> FF
```

2.3.4 Function Calls

MPL allows the programmer to define their own functions. It also has built-in functions such as addition, subtraction, multiplication, divison, equality testing etc. Both the classes of functions can be called in MPL functions or processes. Syntax of a function call is:

$$f \quad (a_1, \ldots, a_n)$$

where f is the function (defined/built-in) which is called with n arguments $a_1, \ldots, a_n$. The function should be called with the right number of parameters otherwise a semantic error is raised.

2.3.5 Variables

Variables allow MPL programs to use variable names in the body of an MPL program. The variable names used in the program should be introduced through a variable pattern. A variable name used inside an MPL program without having being introduced results in a semantic error.

2.3.6 Constants

Constants which can be integers, floats, characters, or strings can be used in the body of an MPL program. An example each of integer, float, character and string are `1, 3.14, 'c'` and `"hello"` respectively.

2.4 Pattern Matching

Like most functional programming languages sequential MPL supports pattern-matching. The syntax

$$v_1, \ldots, v_n \quad \rightarrow t$$

generalizes to

$$\begin{aligned} p_{1,1}, \ldots, p_{1,n} \quad &\rightarrow t_1 \\ &\vdots \\ p_{m,1}, \ldots, p_{m,n} \quad &\rightarrow t_m \end{aligned}$$

which is a list of pattern phrases. A pattern phrase consists of a list of patterns and the corresponding sequential term. When using pattern matching, term t_i is executed only if $p_{i,1}, \ldots, p_{i,n}$ is the first pattern which matches the input.

Variables, constructors, records, and products can be used as patterns giving variable patterns, constructor patterns, record patterns, and product patterns respectively. Sometimes, elements of a pattern aren't used in the corresponding term of the pattern phrase. A special kind of pattern called the don't care pattern, represented by an underscore (`"_"`), is used in such a case.

The remainder of this section gives a few examples of function definitions which use different kinds of patterns in their body. Function `isZero`, defined below, takes an input of type `Nat` and returns `True` if the input is `Zero` and `False` otherwise. `isZero` uses a constructor pattern in the first pattern phrase and a don't care pattern in the second pattern phrase.

```
-- check if the natural number is zero
isZero :: Nat -> Bool =
  Zero -> True
  _    -> False
```

The `or` function, defined below, takes two boolean values as inputs and returns `True` if either of the two inputs is `True` and `False` otherwise. The first pattern matching phrase consists of `True` constructor pattern. The second pattern matching phrase consists of two don't care patterns.

```
-- boolean or function
or :: Bool,Bool -> Bool =
  False,False -> False
  _,_         -> True
```

`or2` function, defined below, rewrites the above defined `or` function with a product pattern of two elements rather than two patterns as was the case in the `or` function definition. Note that a constructor and a don't care patterns have been used inside of a product pattern. A pattern that contains other patterns is called a nested pattern. Also note that in the second pattern matching phrase of the `or2` definition one don't care pattern suffices in comparison to two don't care patterns which are required in the second pattern phrase of the previous `or` function definition.

```
-- boolean or function
or2 :: (Bool,Bool) -> Bool =
```

```
  (False,False) -> False
  _             -> True
```

`thirdProj` function, defined below, takes a record pattern of type `Triple(D,E,F)` and returns the element corresponding to the third destructor from the pattern.

```
-- third projection function
thirdProj :: Triple(D,E,F) -> F =
  (Proj0 := x, Proj1 := y, Proj2 := z) -> z
```

Chapter 3

Concurrent MPL

In this chapter, concurrent MPL programs are described. Concurrent MPL programs are processes, which use protocols and coprotocols. Protocols and coprotocols are the concurrent analogues of sequential MPL's data types and codata types respectively. Processes communicate with each other by sending messages over typed channels.

The syntax for concurrent MPL is essentially new although it follows the ideas of [1]. Protocol and coprotcol declarations are an innovation and refine the syntax ideas in the theses of Subashis [11] and Masuka [10]. The syntax for concurrent MPL evolved significantly during the project. It was decided to allow unconstrained recursive calling of processes rather than just the more disciplined "drives" of earlier conception of the language.

3.1 Channels

In the message passing view of concurrency, channels are important: they act as the conduit for the exchange of messages between processes. An MPL channel connects two processes and messages can be passed in both directions along a channel. A channel connecting two processes must be an output (positive) polarity channel for one process and an input (negative) polarity channel for the other process. Every channel has a *channel type* which governs the legal interactions on that channel. Channel types are dicussed in Section 3.3.

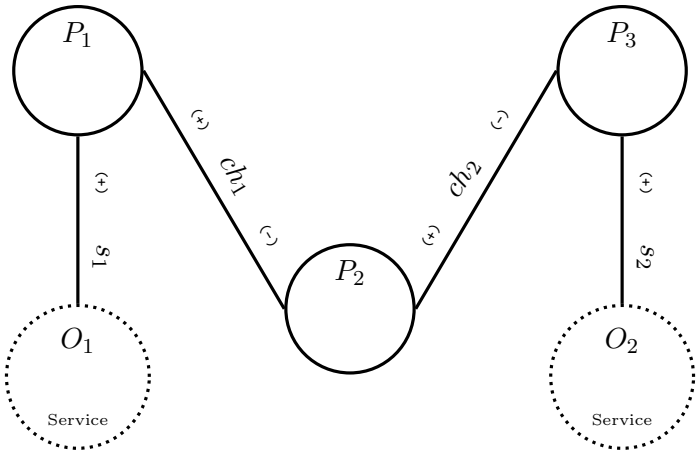

Figure 3.1: Processes Connected by Channels

MPL has some special channels called *service* channels. They are the built-in MPL channels which the programmer can use to interact with the outside world. For example, MPL programs can read user inputs and can display outputs on a terminal using the built-in positive polarity service channels. The console is a special service channel which has negative polarity. Usually, console is the last channel to be closed in a process.

Figure 3.1 illustrates a concurrent MPL program with three processes P_1, P_2 and P_3 represented as circles. A channel ch_1 connects process P_1 with process P_2 and is drawn as a solid line between the two processes. The "+" sign on channel ch_1 towards P_1 and "−" sign towards P_2 indicates that ch_1 acts as an output polarity channel for process P_1 and an input polarity channel for process P_2. Similarly, ch_2 connects processes P_2 and P_3. P_1 also has a positive polarity service channel s_1 which connects it to the outside world.

3.2 Processes

Concurrent MPL programs are organised using processes. An example of an MPL process definition is shown below:

```
proc someProc :: Int | Console (Int) => IntTerm (Int) =
    val | co => s1
```

```
do
    hput PutInt on s1
    put val on s1
    hput Close on s1
    close s1
    hput CloseC on co
    halt co
```

An MPL process is defined with the keyword `proc` followed by the process name, which in the example above is `someProc`. MPL allows the programmer to optionally annotate the processes with a process type. In the example, the type annotation of `someProc` is present in the first line and follows the "`::`" symbol. The process type is `Int | Console(Int) => IntTerm(Int)`. The types on the left side of "`|`" represent the types of the sequential parameters of the process and the right side contains the channel types of the input and the output channels, which are the concurrent parameters of the process. The types on the left side of "`=>`" represent the channel types of the input polarity channels and that on the right side represent the channel types of the output polarity channels. Thus, the process `someProc` has one sequential parameter of the type `Int`, one input polarity channel of channel type `Console(Int)` and one output polarity channel of channel type `IntTerm(Int)`. The actual parameters of `someProc` are given in the second line of the process definition: `val | co => s1`. `val` is the sequential parameter, `co` is the name of the input polarity channel and `s1` is the name of the output polarity channel. The process body is comprised of a sequence of concurrent MPL constructs arranged in a layout format with respect to the keyword `do`.

```
proc p1 :: | Console(Int) => Get(A|TopBot) =
  | co => ch
    do get val on ch
       close ch
       hput CloseC on co
       halt co

proc p2 :: | Put(A|TopBot) => IntTerm(Int) =
  | ch => i1
    do get val on ch
       hput Close on i1
       close i1
       halt ch

run :: Console(Int) => IntTerm(Int) =
  console => intTerm1
    do plug
        p1 ( | console => ch)
        p2 ( | ch => intTerm1)
```

Table 3.1: Example:Achieving Type Safety in MPL

3.3 Channel Types

MPL brings a notion of type safety to the concurrent world. *Channel types* are central to this notion of type safety. A channel type describes the permissible actions on a channel. MPL allows the programmers to define complex user defined channel types called protocols or coprotocols constructed from the basic built-in channel types.

Section 3.3.1 discusses how channel types bring type safety to the concurrent world, Section 3.3.2 discusses the built-in channel types in concurrent MPL and Section 3.3.3 describes protocol and coprotocol definitions.

3.3.1 Achieving Type Safety Using Channel Types

Consider the example of the program in Table 3.1 consisting of processes `p1`, `p2`, and the main process. Process `p1` and `p2` are connected via a channel `ch` using `plug` construct in the main process. Both `p1` and `p2` try to read a value from the channel `ch`. The diagrammatic

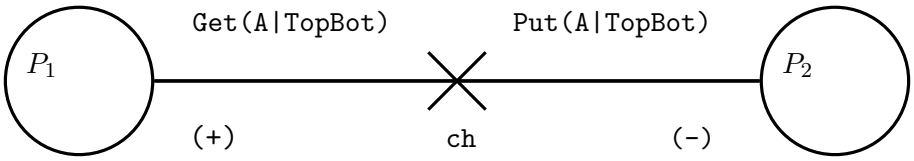

Figure 3.2: Protocol Error in MPL Processes

representation of the program is shown in Figure 3.2 where circles P_1 and P_2 represent processes `p1` and `p2` respectively. In the program, both the processes try to get a value on the channel `ch`. `get` and `put` are MPL constructs that fetch a value from a channel and put a value on a channel respectively. The `close` construct closes a channel.

Intutitively, the configuration in Figure 3.2 seems wrong because both the processes can't simulatenously get a value on a channel. If one process puts a value on a channel then the other process must read that value. Thus, a configuration like the one in Figure 3.2 should result in a type error.

MPL infers the channel type of `ch` from the two processes it is connecting and checks if the two channel types match. If the channels types don't match then a type error is reported. In order to infer the channel type of channel *ch*, the built-in channel types `Get, Put` and `TopBot` are used. This is not an exhaustive set of base protocols: the set of all the built-in channel types are provided in section 3.3.2.

On an output polarity channel, the `get` construct results in a `Get` channel type and the put construct results in a `Put` channel type. However, on an input polarity channel, `get` results in a `Put` channel type and `put` results in a `Get`. This swap is done so that the channel types obtained from the processes on the opposite polarities of a channel match. Closing a channel of any polarity results in the protocol `TopBot`. On inferring the channel type of `ch` from processes P_1 and P_2, `Get(A|TopBot)` and `Put(A|TopBot)` are obtained respectively. Since, the two channel types don't agree, a type error results.

3.3.2 Built-in Channel Types

All the channel types in MPL are constructed using the following built-in channel types.

`Get(A|P)` - `Get` channel type takes a sequential type `A` and a channel type `P`.

`Put(A|P)` - `Put` channel type takes a sequential type `A` and a channel type `P`.

`P1 (*) P2` - The symbol "`(*)`" represents the built-in channel type *tensor*. Tensor is normally represented by the symbol $\otimes$ in Mathematics. Tensor channel type takes two channel types `P1` and `P2`.

`P1 (+) P2` - The symbol "`(+)`" represents the built-in channel type *par*. Par is normally represented by the symbol $\oplus$ or $\parr$ in Mathematics. Par channel type takes two channel types `P1` and `P2`.

`TopBot` - `TopBot`, which is an acronym for `Top` and `Bottom` types, is a built-in channel type and it doesn't take any arguments.

`Neg(A)` - `Neg`, standing for negation, takes one argument. Table 3.7 shows an example of a program that uses the `Neg` protocol.

3.3.3 Defining (Co)Protocols in MPL

Concurrent MPL allows the programmer to define complex channel types called protocols or coprotocols using the built-in channel types.

Protocols

An example of a protocol definition, which is also the channel type of the service channels `intTerm1`,`intTerm2`,...with `A = Int`, is:

```
protocol IntTerm (A) => P =
      GetInt :: Get (A|P) => P
```

```
PutInt :: Put (A|P) => P
Close  :: TopBot    => P
```

A protocol is defined using the keyword `protocol`. The name of the defined protocol is `IntTerm` which is polymorphic in type `A`. `P` is the state variable. The syntax `IntTerm(A) => P` follows the syntax of a sequential data type and is used to indicate that a protocol is an initial algebra of the concurrent world. The protocol `IntTerm` has three handles, `GetInt`, `PutInt` and `Close`: handles are the analogues of constructors in the sequential world. The type of a handle is obtained by substituting the state variable `P` with the protocol name `IntTerm(A)` in the type expression corresponding to that handle in the protocol definition. For example, the type of the handle `GetInt` is given by:

```
(Get(A|P) => P) [IntTerm(A)/P]
```

If the type of a channel is `IntTerm` then it means that any of the three handles of `IntTerm` can be put on the channel. Once a handle has been put on a channel, the only actions allowed on the channel are the actions permitted by the channel type of that handle. Note that the handles `GetInt` and `PutInt` are recursive.

Coprotocols

An example of a coprotocol definition is:

```
coprotocol CP => Console (A) =
    GetIntC   :: CP => Put (A|CP)
    PutIntC   :: CP => Get (A|CP)
    CloseC    :: CP => TopBot
```

A coprotocol is defined using the keyword `coprotocol`. The name of the defined coprotocol is `console`, which is polymorphic in the type variable `A`. Coprotocols are the codata types

of the concurrent world. The coprotocol `Console` has three cohandles `GetIntC`, `PutIntC` and `CloseC`. The type of a cohandle is given by subtituting the state variable with the coprotocol name. The type of the cohandle `GetIntC` is given by:

```
(CP => Put (A|CP)) [Console(A)/CP]
```

Notice that the difference between protocols and coprotocols is less dramatic than that between data and codata as the concurrent world is more symmetric.

3.4 Concurrent MPL constructs

The body of MPL's processes are written using concurrent MPL constructs. These are plug, id, neg, get and put, hput and hcase, split and fork and close and halt.

Most concurrent MPL constructs perform an action on a channel. In MPL concurrency is modeled using message passing between processes, so it is only logical that most concurrent constructs should deal with channels.

With the exception of process call, id, neg, and plug, all concurrent MPL constructs come in pairs. The constructs of a pair are dual to each other, i.e they work on the opposite polarities of a channel. This is because in the message passing view of concurrency, for a process to respond to an action over a channel, another process has to drive that action on the other end of the channel and vice versa. Thus, in each pair of constructs mentioned above one is a driver construct and the other is a reaction construct. For example, for a process to get a value on a channel, some other process should have put a value on the channel. Thus, get is a reaction construct driven by the put construct.

A brief description of all the concurrent MPL constructs is provided in Table 3.2. In the next few sections (Section 3.4.1 to Section 3.4.7), the concurrent MPL constructs are described in details.

process call	calls a defined process
id	equates two channels
neg	negates a channel
plug	connects two processes by a channel
get	gets a value on a channel
put	puts a value on a channel
hput	puts a handle on a channel
hcase	cases on the handles obtained on channel
split	splits a channel into two channels
fork	forks two new processes
close	closes a channel
halt	closes a channel. Usually the last channel is halted.

Table 3.2: Concurrent MPL Constructs

```
proc p1 :: Int | Console(Int) => IntTerm(Int) =
    val | co => s1
      do hput PutInt on s1
         put val on s1
         hput Close on s1
         close s1
         hput CloseC on co
         halt co

run :: Console(Int) => IntTerm(Int) =
  console => intTerm2
    do p1 ( 5 | console => intTerm1 )
```

Table 3.3: Example : Process Call

3.4.1 Process Call

In MPL, a process which has been defined can be called inside another process as the last command in the process or as part of a plug command.

Table 3.3 shows an example of an MPL program which uses a process call. The output of the program is the integer five printed on the output polarity service channel, `intTerm1`.

Program uses the protocol `IntTerm` and the coprotocol `Console` for the output and input service channels respectively. `IntTerm` and `Console` were defined in Section 3.3.3. Main process calls process `p1` with 3 arguments: one sequential argument (integer five) and

two channels (`console` and `intTerm1`) which are the concurrent arguments.

3.4.2 The plug Construct

The plug construct allows one to connect two or more processes via channels. A plug command can only occur as the last command in a process block. Once the two processes are plugged together, they can exchange messages.

The syntax of the plug construct is illustrated by:

```
plug
    p1 ( | ich => ch)
    p2 ( | ch => och)
```

where the keyword plug is followed by the processes being plugged. Here processes `p1` and `p2` are plugged together on channel `ch`, which acts as an output polarity channel for `p1` and an input polarity channel for `p2`. For the plug construct to type check, the channel types of the plugged channel inferred from the two processes it connects must unify.

One can also plug multiple processes using the plug construct as can be seen in the example below where channel `ch1` is plugged between processes `p1` and `p2` and channel `ch2` is plugged between processes `p2` and `p3`.

```
plug
    p1 ( | ich => ch1)
    p2 ( | ch1 => ch2)
    p3 ( | ch2 => och)
```

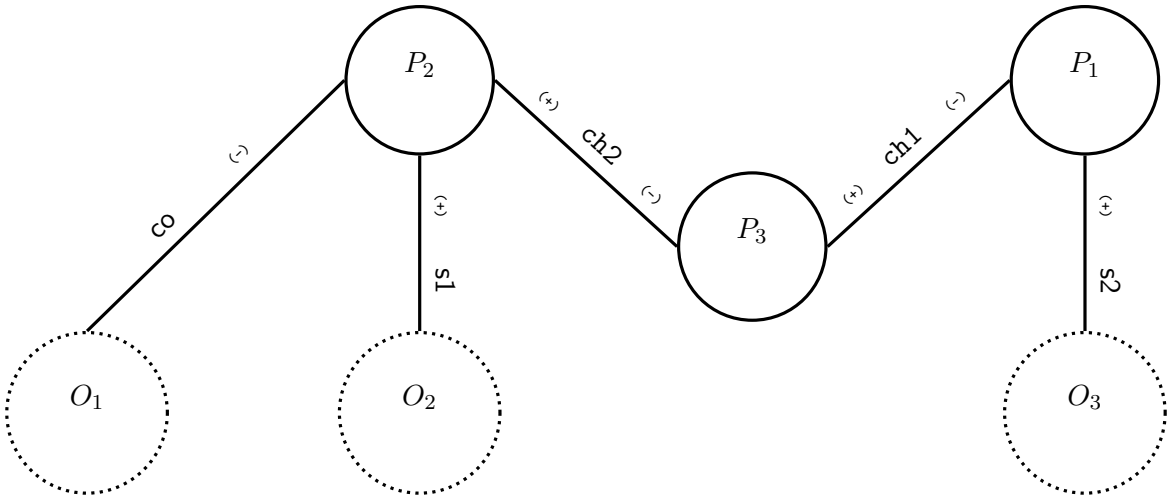

Figure 3.3: The get and put Example

3.4.3 The get and the put Constructs

The get and the put are basic MPL constructs. The get construct reads a value on a channel. Its syntax is:

$$\mathsf{get}\ \ v\ \ \mathsf{on}\ \ \alpha$$

where v is a variable and α is a channel name. The value read from the channel α is bound to the variable v. The get construct adds the variable v to sequential context of the process in which it is used. Thus, v is visible to the subsequent MPL commands. The put construct puts a sequential term on a channel. Its syntax is:

$$\mathsf{put}\ \ t\ \ \mathsf{on}\ \ \alpha$$

where t is a sequential term which is put on a channel α. The typing of the get and the put constructs was discussed at the end of Section 3.1.

Table 3.4 shows an example of an MPL program that uses the get and the put constructs. Figure 3.3 shows a diagrammatic representation of the program which is used to give a

```
proc p2 :: | Console(Int) => IntTerm(Int), Put(Int|TopBot) =
    | co => s1,ch2
      do hput GetInt on s1
         get x on s1
         put x on ch2
         hput Close on s1
         close s1
         close ch2
         hput CloseC co
         halt co


proc p3 :: |  Put(Int|TopBot) => Put(Int|TopBot) =
    | ch2 => ch1
      do get x on ch2
         put x on ch1
         close ch2
         halt ch1

proc p1 :: | Put(Int|TopBot) => IntTerm(Int) =
     | ch1 => s2
        do get x on ch1
           hput PutInt on s2
           put x on s2
           hput Close on s2
           close s2
           halt ch1

run :: Console(Int) => IntTerm(Int),IntTerm(Int) =
  console => intTerm1,intTerm2
    do plug
         p2 ( |console => intTerm1,ch2)
         p3 ( | ch2 => ch1)
         p1 ( | ch1 => intTerm2 )
```

Table 3.4: Example : The get and the put Constructs

description of the program.

In the example program in Table 3.4 there are three processes, `p1`, `p2`, and `p3` apart from the main process. In the main process, the three processes mentioned above are plugged together. In the Figure 3.3, the three processes `p1`, `p2`, and `p3` are represented as three circles named P_1, P_2, and P_3 respectively.

When the program is executed, a terminal (output polarity service channel `intTerm1`) pops up asking for a number to be entered. Once the user has entered a number, another terminal (output polarity service channel `intTerm2`) pops up displaying this number.

Process P_2 asks for a value on its service channel `s1`. P_2 then puts this value on the channel `ch2` from where P_3 reads it. P_3 then puts the value it has read on the channel `ch1` from where P_1 reads it. P_1 then puts the value it has read on its service channel `s2`. This value is now visible to the outside world.

Executing the get Before the put

The program in Table 3.4 is simple to understand if one considers that the processes are scheduled in the order P_2, P_3, and P_1 which means that the process P_2 gets a value and passes it to the process P_3 which in turn passes to the process P_1. However, the concurrent semantics for MPL allows the processes to be scheduled in any order. For example, the order P_1, P_2, and P_3 is also a valid schedule of the above processes. In the second schedule the process P_1 tries to get a value on the channel `ch1` before the process P_3 puts a value on that channel. Similarly, process P_3 tries to get a value on the channel `ch2` before the process P_2 puts a value on the channel.

MPL handles the situation described above, where a process is trying to get a value from a channel before that value has been put on the channel, by putting the process requesting the get operation to sleep and removing it from the set of active processes. Once a value is put on the channel, the process requesting get is woken up and put in the set of active processes again, from where it can be scheduled. This mechanism is used not only with the

get and the put constructs but also with other concurrent constructs which come in pairs like the split and the fork, and the hput and the hcase in the situations where the process containing the reaction construct happens before the process containing the driver construct.

3.4.4 The id and the neg Constructs

The id construct equates two channels. The syntax `ch2 |=| ch1` means that the channel `ch1` is equated with the channel `ch2` which means the following:

- The messages written on the channel `ch1` are now available on the channel `ch2`, i.e one can access the messages available on the channel `ch1` from the channel `ch2`.
- The messages to be sent via `ch1` can now be sent via `ch2`, i.e. one can write a message on `ch2` and have it available on `ch1`.

When equating two channels of the same polarity, the neg construct is used to change the polarity of one of the channels ("bending" the wire). For example, if both `ch1` and `ch2` are of the same polarity then they can be equated by `ch2 |=| neg(ch1)`. If the type of the channel `ch1` is `A`, then the type of the channel `ch2` must be `Neg(A)`. A significant example of the id and neg constructs is provided in Section 3.4.8.

3.4.5 The hput and the hcase Constructs

The hput construct puts a (co)handle of a (co)protocol on a channel. Its syntax is:

$$\mathsf{hput}\ H\ \mathsf{on}\ \alpha$$

where H is the handle or the cohandle being put on the channel α. The hcase construct can be thought of as the concurrent counterpart of the case construct from the sequential MPL. The case construct branches on the constructors of a data type and executes a sequential construct corresponding to the data constructor selected from the branch. Similarly, the

hcase construct branches on the handles of a protocol and runs a sequence of concurrent commands corresponding to the selected handle from the branch. Its syntax is:

$$
\begin{array}{l}
\textsf{hcase } \alpha \textsf{ of} \\
\quad \begin{array}{lcl}
H_1 & \rightarrow & c_{11}; \ldots; c_{1a} \\
\vdots & & \vdots \\
H_m & \rightarrow & c_{m1}; \ldots; c_{mn}
\end{array}
\end{array}
$$

where $H_1, \ldots, H_n$ are the (co)handles of a (co)protocol, α is the channel being hcased on, and $c_{i1}, \ldots, c_{ij}$ represent the j concurrent MPL commands which are executed if the (co)handle H_i is selected.

The hput and the hcase are dual to each other: a handle is hput on one end of a channel - the driver - and reacted to by an hcase at the other end. For a protocol, one hputs a handle on an output polarity channel and hcases on the handles on an input polarity channel. Dually, for a coprotocol, one hputs a cohandle on an input polarity channel and hcases on the cohandles on an output polarity channel.

Table 3.5 shows an example of a program that uses the hput and the hcase constructs. The processes of the program along with their channels are shown in Figure 3.4. The program consists of two processes P_1 and P_2 which are plugged along a channel `ch`. Process P_1 reads an integer value on the channel `i1` and checks if that value is greater than ten. If the integer value is greater than ten then P_1 hputs the handle `PutInt` on the channel ch, puts the value on the channel `ch`, hputs a handle `GetInt` on the channel, receives an integer value on the channel `ch`, prints this received integer value on the channel `intTerm1`, and calls itself again. If the value is less than ten, then P_1 terminates. Process P_2 hcases on the handles obtained on the channel `ch` and performs the following actions for the various handles:

`GetInt`: The process reads a value from its service channel `intTerm2` and puts it on the

```
proc p1 :: |Console(Int) => IntTerm(Int),IntTerm(Int)=
    | co => i1,ch
     do hput GetInt on i1
        get x on i1
        case x > 10 of
          True
            do  hput PutInt on ch
                put x on ch
                hput GetInt on ch
                get y on ch
                hput PutInt on i1
                put y on i1
                p1(|co => i1,ch)

          False
            do hput Close on i1
               close i1
               hput Close on ch
               close ch
               hput CloseC on co
               halt co

proc p2 ::  | IntTerm(Int) => IntTerm(Int)=
    | ch => i2
      do  hcase ch of
            GetInt
              do hput GetInt on i2
                 get x on i2
                 put x on ch
                 p2 ( | ch => i2)
            PutInt do
              do get x on ch
                 hput PutInt on i2
                 put x on i2
                 p2 ( | ch => i2)
            Close
              do hput Close on i2
                 close i2
                 halt ch

run :: Console(Int) => IntTerm(Int),IntTerm(Int) =
  console => intTerm1,intTerm2
    do plug
         p1 ( | console => ch,intTerm1)
         p2 ( | ch => intTerm2)
```

Table 3.5: Example : The hput and the hcase constructs

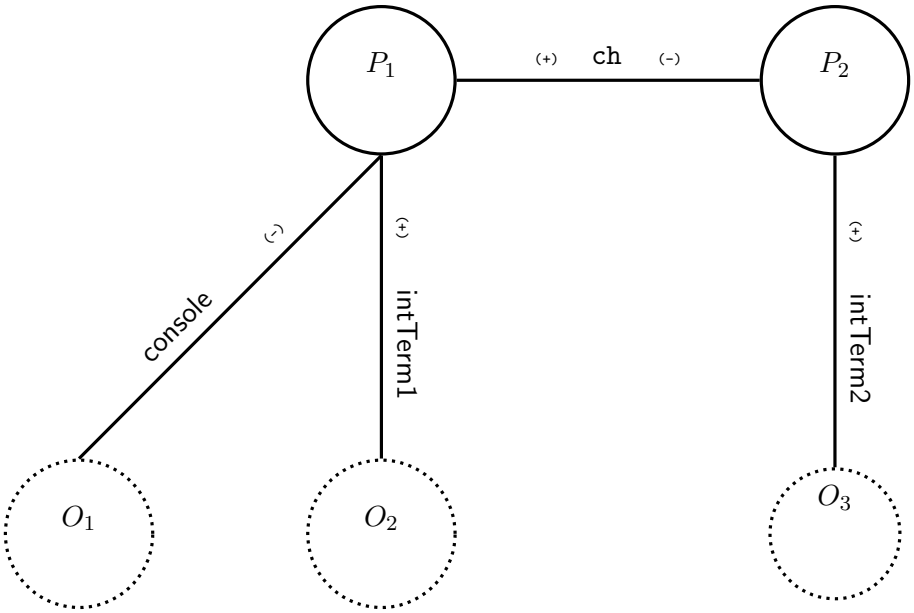

Figure 3.4: The hput and hcase Example

channel *ch*.

`PutInt`: The process reads a value from the channel `ch` and puts it on the channel `intTerm2`.

`Close`: The channels are closed and the process is terminated.

3.4.6 The split and the fork Constructs

Creating/forking new processes is a requirement in many concurrent programs. In MPL, creation of the new processes is achieved using the fork construct which divides a process into two new processes. Consider a scenario where two processes are connected via a channel and one of these two processes forks into two new processes. In such a case, the channel that connects the two process also needs to be split. The split construct splits a channel into two channels. The split and the fork constructs are dual to each other. The syntax of the fork construct is:

fork α of

$$\alpha_1 \rightarrow p_1$$

$$\alpha_2 \rightarrow p_2$$

where α indicates the channel that should be split into α_1 and α_2 so that the processes can successfully be forked. p_1 and p_2 are the two forked processes. Channel α_1 is associated with the first forked process. The channel α_2 is associated with the second forked process. When a process is forked, the set of channels of the process are distributed between the two processes such that the two sets of channels for the processes are disjoint and their union results in the original set of channels.

Once a channel is forked, the reaction should be a split. The syntax of the split construct is:

$$\textsf{split} \;\; \alpha \;\; \textsf{into} \;\; \alpha_1, \alpha_2$$

where channel α represents the channel which is split into two new channels α_1 and α_2. Once a channel is split, it is no longer visible to the subsequent concurrent MPL constructs in the process block. However, the newly created channels are visible and can be used by the subsequent constructs.

Splitting an input polarity channel or forking on an output polarity channel results in the channel type `(*)` (tensor). Splitting an output polarity channel or forking on an input polarity channel results in the channel type `(+)`(par).

Table 3.6 shows an example of a program that uses the split and the fork constructs. The configurations of the processes before and after the split and the fork constructs are executed are shown through the two diagrams in Figure 3.5. The first diagram represents the processes and channels in the program before the split and the fork constructs are executed and the second diagram represents the processes and the channels once the constructs are executed.

The main process plugs the channel `ch` between the processes P_1 and P_2. P_1 gets a value

```
proc p1 :: |Console(Int) =>
 Put(Int|TopBot)(+)Put(Int|TopBot)
 ,IntTerm(Int) =
  | console => ch,intTerm1
    do hput GetInt on intTerm1
       get x on intTerm1
       split ch into ch1,ch2
       put x on ch1
       put x on ch2
       close ch1
       close ch2
       hput Close on intTerm1
       close intTerm1
       hput CloseC on console
       halt console

proc p21 ::| Put (Int|TopBot) =>
             IntTerm(Int) =
    | ch1 => intTerm2
      do get x on ch1
         hput PutInt on intTerm2
         put x*x on intTerm2
         hput Close on intTerm2
         close intTerm2
         halt ch1
```

```
proc p22 :: | Put(Int|TopBot) =>
              IntTerm(Int) =
    | ch2 => intTerm3
      do get x on ch2
         hput PutInt on intTerm3
         put x*x*x on intTerm3
         hput Close on intTerm3
         close intTerm3
         halt ch2

proc p2::|Put(Int|TopBot)(+)Put(Int|TopBot)=>
          IntTerm(Int),IntTerm(Int) =
    | ch => intTerm2,intTerm3
      do fork ch as
           ch1 do
             p21 ( | ch1 => intTerm2)
           ch2 do
             p22 ( | ch2 => intTerm3)

run :: Console(Int) =>
    IntTerm(Int),IntTerm(Int),IntTerm(Int) =
 console => intTerm1,intTerm2,intTerm3
    do plug
         p1 ( | console => ch,intTerm1)
         p2 ( | ch => intTerm2,intTerm3)
```

Table 3.6: Example : The split and the fork constructs

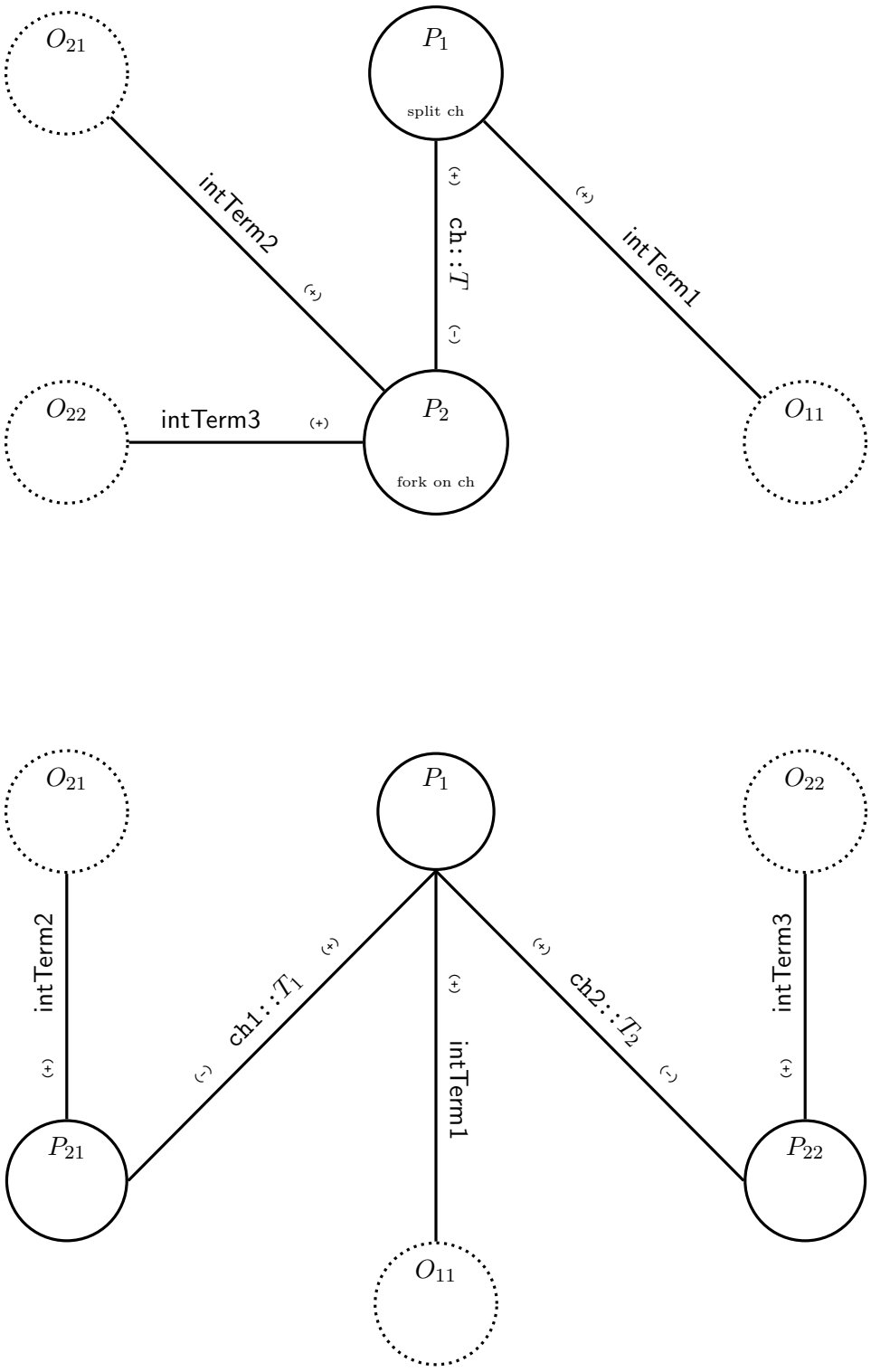

Figure 3.5: The split and fork Example $(T = T_1\ (+)\ T_2)$

on its service channel `intTerm1`, splits its output polarity channel `ch` into two channels `ch1` and `ch2`, and puts the obtained value from the service channel on `ch1` and `ch2`. The process P_2 forks two new processes P_{21} and P_{22} on the channels `ch1` and `ch2` respectively. The process P_{21} gets the value that was put on the channel `ch1`, squares it and puts the squared value on its service channel `intTerm2`. Similarly, the process P_{22} gets the value that was put on the channel `ch2`, cubes it and puts the cubed value on its service channel `intTerm3`.

3.4.7 The close and the halt Constructs

The close and the halt constructs are used to close a channel. All the channels except the last channel of a process are closed: the last channel is halted signifying the halting of the process.

3.4.8 Memory Cell

This section gives an example of an MPL program that uses most of the concurrent MPL constructs along with the Neg protocol. The example illustrates how the topology of the communicating processes in MPL can be changed dynamically. To illustrate mobility, we shall show how a mutable memory cell can be passed between two processes. The MPL program for the memory cell is shown in Table 3.7 and the corresponding configurations of processes are shown in Figure 3.6 and 3.7.

The program consists of three processes: `p1`, `p2`, and `memory`. These processes are represented by the circles $\mathtt{P}_1$, $\mathtt{P}_2$ and `Memory` in Figure 3.6. The memory cell, represented by the process `memory`, can remember a value that is passed to it, can provide, on demand, the value it is remembering, or can be closed. A memory cell communicates to the outside world through an input polarity single channel: through which it receives values and outputs remembered values. To program these actions, the protocol `MEM`, with handles `PUT`, `GET`, and `CLS`, is defined which acts an interface to the memory cell. For the handle `GET`, the memory cell receives a value on its channel `ch` and remembers it. For the handle `PUT`, the memory

cell puts the remembered value on the channel `ch` and still remembers the same value. For the handle `CLS` memory cell halts.

The program mobilizes the memory cell, `Memory`, between the two processes P_1 and P_2 and the memory cell communicates alternately with the two processes P_1 and P_2. The transfer of the memory cell between the two processes is acheived using the protocol `Passer`. The `Passer` protocol is put on the output channel `p` of the process P_1. The process P_1 then gets the value `y` from the memory cell, displays this value on its output polarity service channel `io`, receives a value `x` on `io`, and changes the value stored in the memory cell to `x`.

The process P_2 has `p` as its input channel and it hcases on the handle of the `Passer` protocol. It gets a value `y` from the memory cell and displays it on its output polarity service channel `in`. P_2 then receives a value `x` on `in` and updates the memory cell with that value.

To begin with, the memory cell is attached to the process P_2 on the channel `mem` and process P_2 is connected to process P_1 on the channel `p` as shown by the first diagram in Figure 3.6.

In order to transfer the memory cell to the process P_1, the channel `p` is divided into three channels: `mm`, `nm`, and `pp` as shown in the second diagram of Figure 3.6. Then the channel `mem`, the interface channel of the memory cell, is equated with the channel `mm` and the channel `pp` is negated and supplied as the input channel in the recursive call for process P_2. In this configuration, shown by the third diagram in Figure 3.6, memory cell is attached to process P_1 via channel `mm`.

In the recursive call for process P_1, `mm` is negated and equated with the `nm` and `pp` is used in the recursive process call as an output channel. Equating the two channels reattaches the process `Memory` to the process P_2. This is shown in Figure 3.7. In this way, the memory cell is passed between the two processes.

```
protocol MEM (A) => P =
  PUT :: Put (A|P) => P
  GET :: Get (A|P) => P
  CLS :: TopBot    => P

protocol Passer(A) => P =
  Pass :: A (+) (Neg(A) (*) P) => P

proc memory :: Int | MEM(Int) =>  =
  x | ch => -> do
      hcase ch of
        PUT
          do get y on ch
             memory(y|ch => )
        GET
          do put x on  ch
             memory(x|ch => )
        CLS
          do halt ch

proc p2 :: | Passer(MEM(Int)) =>
              IntTerm(Int),MEM(Int) =
  | p => in, mem -> do
    hcase p of
      Pass
        do
          hput GET on mem
          get y on mem
          hput PutInt on in
          put y on  in
          hput GetInt on in
          get x on  in
          hput PUT on mem
          put x on  mem
          fork p as
            mm
              do mm |=| mem
            nmpp
              do split nmpp into nm,pp
                 plug
                    p2 ( | pp => in,z)
                    z |=| neg nm
```

```
proc p1 :: | => Passer(MEM(Int)),
                IntTerm(Int) =
     | => p,io
    do
      hput Pass on p
      split p into mm,nmpp
      hput GET on mm
      get y on  mm
      hput PutInt on io
      put y on io
      hput GetInt on io
      get x on io
      hput PUT on mm
      put x on  mm
      fork nmpp  as
        nm
          do nm |=| neg mm
        pp
          do p1( | => pp,io)


run :: =>IntTerm(Int),IntTerm(Int)=
    => intTerm1,intTerm2
     do plug
          p1( | => p,intTerm1)
          p2( | p => intTerm2,mem)
          memory(10 |mem => )
```

Table 3.7: Memory Cell: The id and the neg constructs

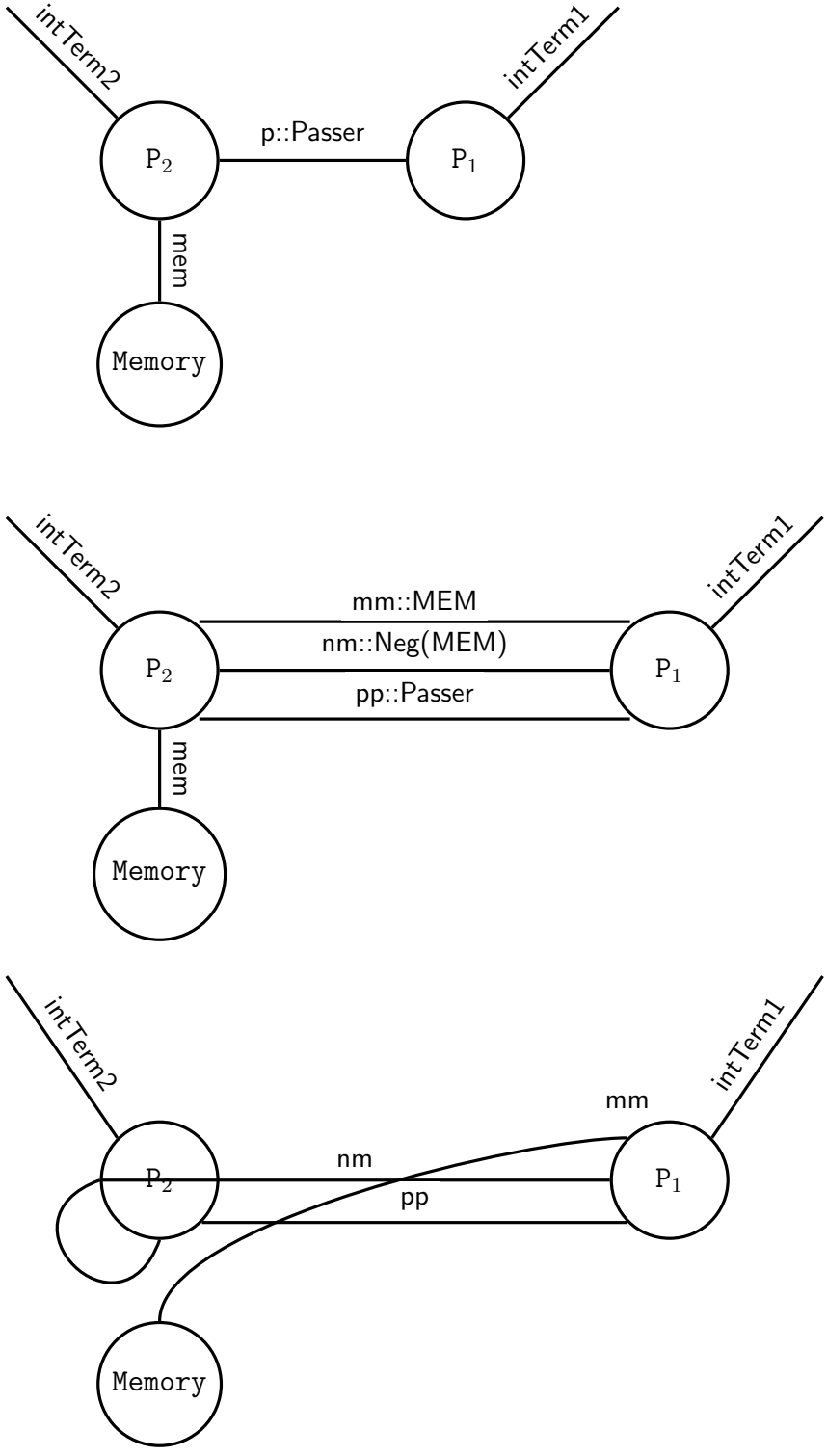

Figure 3.6: Memory Cell: The id and neg Example

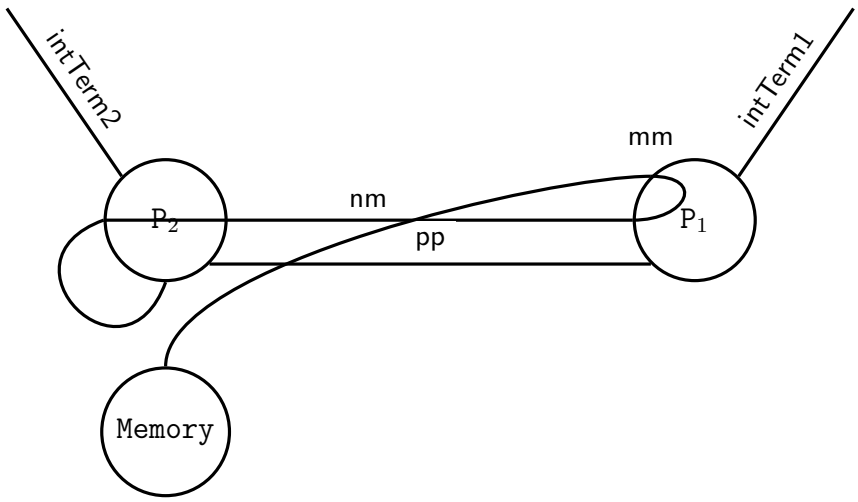

Figure 3.7: Memory Cell: The id and neg Example

Chapter 4

Type Inference of MPL Programs

Once an MPL program is lexed and parsed, an abstract syntax tree (AST) is generated for the program. An AST is a faithful representation of the original MPL program in that all the important information present in the original program is present in the AST. The next stage in the interpretation of the MPL program is type inferencing. This step ensures that only type safe programs continue the process of interpretation. This chapter deals with the type inference process of the various elements of an MPL programs like functions, terms, patterns, pattern phrases, processes, process phrases, and process commands.

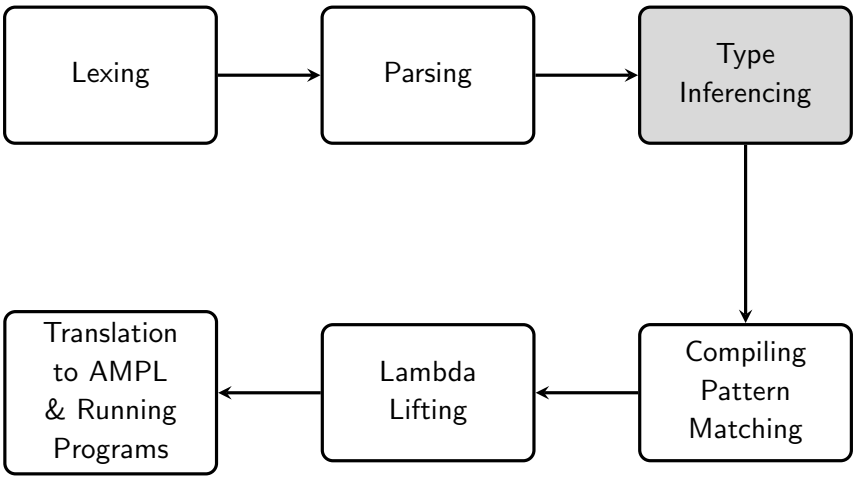

Figure 4.1: Interpretation Stages of MPL

The type inference process described in this chapter follows the mechanism outlined in [5].

The type inference process of MPL programs involves two main steps:

- **Generating Type Equations** - Type Equations represent the constraint between the different parts of the construct being type inferred. Using the typing rules described in this chapter, the proof tree of a term is annotated with type equations in the proof search direction. The formal definitions of the type equations and their Haskell data type representation are discussed in Section 4.1 and the generation of type equations for the sequential and concurrent structures of MPL programs are discussed in Chapter 5 and Chapter 6 respectively. Type equation generation step uses a data structure called *Symbol Table* which acts as a storage of the symbols used in a programs. A description of the role of the symbol table in the type inference process in Section 4.3.

- **Solving Type Equations** - The type equations are solved in order to get the most general type of the functions, and processes in an MPL program. The solution of the type equations generated for MPL programs is discussed in the Section 4.2.

Not only does the general type solving algorithm allows for better location of type errors but it also allows for universal and existential qunatification to be handled. The universal quantification is used to ensure matching to user specified types. The generation of type equations (described in Chapter 5 and 6), which had to be defined for MPL, followed the ideas in the course notes [5]. The type inference for mutually recursive functions, the fold, and unfold were amongst the more challenging aspects of the thesis.

4.1 Type Equations

In our type inference algorithm, a type equation is inductively defined as:

$$\texttt{Type Equation} := \begin{cases} \texttt{(Type Variable = Type Expression)} \\ \exists\ v_1, \ldots, v_n\ .\ \{\texttt{Type Equation}\} \\ \forall\ u_1, \ldots, u_m\ .\ \{\texttt{Type Equation}\} \end{cases}$$

A type equation is either a type variable assigned to a type expression, a set of type equations having existential type binders, or a set of type equations having universal type binders. The existential type binders are the set of variables $v_1, \ldots, v_n$, which are associated with the $\exists$ symbol. The universal type binders are the set of variables $u_1, \ldots, u_m$, which are associated with the $\forall$ symbol. The notations $\forall$, and $\exists$ should just be viewed as the syntax for type binders and not as the universal and existential quantification of predicate logic.

The above definiton of type equation can be expressed by the Haskell data type `TypeEqn` shown in the Figure 4.2. A description of the `TypeEqn` data type is:

- The constructor `TSimp` is the non-recursive phrase of the definition. The recursive branch contains the existential and universal type binders is represented by the constructor `TQuant`.

- The `TSimp` constructor has as argument a pair of type expressions which are to be made equal. The type expression are represented by the Haskell data type `Type`, which is a data type defining the valid MPL types, both concurrent and sequential.

- The first argument of the constructor `TExist`, `[ExistVar]`, is a list of existentially bound variables. `ExistVar` is type synonymous to an integer. The second argument of `TExist` is a list of type equations to be satisfied. Thus, `TypeEqn` is a recursive type.

- The first argument of the constructor `TUniv`, `[ExistVar]`, is a list of universally bound variables. `UnivVar` is type synonymous to an integer. The second argument of `TUniv` is a list of type equations to be satisfied.

4.2 Solving Type Equations

Type inference generates a list of type equations with the universal and existential variable bindings and when one solves the equations generated by a type inference algorithm it is worth using the extra information inherent in the variable bindings: this information

```
type Name       = String

-- Haskell data type for Type expression in MPL
data Type =    Unit
             | DataType (Name,[Type])
             | CodataType (Name,[Type])
             | Prod  [Type]
             | Const ConstType
             | TVar Int
             | Fun ([Type],Type) -- function type
             | Get (String,Type)
             | Put (TypeExpr,Type)
             | Neg Type
             | TopBot
             | Protocol  (Name,[Type])
             | Coprotocol(Name,[Type])
             | Tensor(Type,Type)
             | Par   (Type,Type)
             | Proc ([Type],[Type],[Type]) -- process type

-- constant types in MPL
data ConstType =   ConstInt
                 | ConstChar
                 | ConstDouble
                 | ConstString

type Var        = Int  -- variables are represented as integers
type UnivVar    = Var  -- universally bound variables
type ExistVars  = Var  -- existentially bound variables

-- Haskell data type for type equations
data TypeEqn =   TSimp  (Type,Type)
               | TExist [ExistVar] [TypeEqn]
               | TUniv  [UnivVar]  [TypeEqn]
```

Figure 4.2: Haskell Data Type for Type Equation and MPL Type Expressions

not only simplifies the problem but in a real implementation can help one to locate where the typing went wrong (if it did). The modified type inference algorithm described here works inductively by trying to eliminate the universally and existentially bound variables; by associating with each universally and existentially bound variable the line number (and term) at which the equation collection introduced the variable, that area of the code can be identified as the source of failure. Note that, in general, the cause of a type error is hard to determine as it may not even be a local problem: this makes error reporting difficult. The above approach has the merit of localising the problem as far as possible and is straight forward to implement. Type inference produces either a type failure or the most general type of the MPL constructs. The objective of the type inference process is to return either a type failure or a *package* of type equations.

We may express what must be done to solve these equations in various ways: one method is to give a set of rules for simplifying type equations. These rules listed below are described with the help of the functions check, match, and coalesce, which are defined in the Table 4.3 and described in section 4.2.1:

1. Any occurrence of an equation of the form $x \equiv t$ where $x \in t$ and $x \neq t$ causes failure (*occurs check* failure). This is implemented by the function check.

2. An occurence of an equation of the form

$$F(t_1, \ldots, t_n) \;\equiv\; G(t'_1, \ldots, t'_m)$$

where F and G are either function names, data names, codata names, protocol names, or coprotocol names, and $F \neq G$ or $n \neq m$ results in match failure. An occurence of an equation of the form

$$F(t_1, \ldots, t_n) \;\equiv\; F(t'_1, \ldots, t'_n)$$

can be replaced by the equations $t_1 \equiv t'_1, \ldots, t_n \equiv t'_n$ (matching). Match failure and matching are achieved using the match function.

3. Universally and existentially quantified empty lists of equations, $\forall x.()$ and $\exists x.()$, can be removed.

4. Existentially bound variables, for which there is a substitution, can be eliminated by substituting: $\exists x.(x \equiv t, E)$ can be rewritten as $E[t/x]$.

5. Universally bound variables in an equation, $\forall x.E$, can be removed if there is either a trivial equation of the form $x \equiv x$ in E, or if x doesn't occur in E. If there is an equation of the form $x \equiv t$ and $x \neq t$ then a type error results.

6. Scope manipulations for the existentially bound variables:

$$\exists x.\exists y.E = \exists y.\exists x.E \qquad\qquad E_1, \exists y.E_2 = \exists y.E_1, E_2 \quad y \notin E_1$$

7. Scope manipulations for the universally bound variables:

$$\forall x.\forall y.E = \forall y.\forall x.E \qquad\qquad E_1, \forall y.E_2 = \forall y.E_1, E_2 \quad y \notin E_1$$

The above rules can be translated into an equation solving algorithm: it starts at the leaves of the tree of type equations and works down towards the root, (i.e. it is a fold). The most general type of MPL constructs are obtained via an intermediate data structure called a *package*. The type equations corresponding to an MPL construct is converted into a package which is then continuosly reduced till its normal form is reached. The normal form of the package gives the most general type of the MPL construct. A package is a four tuple with type:

```
([Var],[UnivVar],[ExistVar],[(Var,Type)])
```

where:

- The first element of the tuple, `Vars`, is a list of free variables whose substitutions one is trying to determine.
- The second and the third element of the tuple are lists of universal and exisential bound variables respectively which could not be eliminated.
- The fourth argument is a list of substitution for the free variables. The variables not mentioned are assumed to be substituted by the identity substitution x/x.

A description of the algorithm for solving type equations is given below:

1. Given a basic type equation without quantified variables $t_1 \equiv t_2$, the type expressions t_1 and t_2 must be matched to obtain a list of substitutions together with the list of free variables, and empty universally, and existentially bound variable lists.
2. Given an existentially or universally bound empty list of type equations, the package `([],[],[],[])` is returned, i.e. the free variable list, the universally, and existentially bound variable list, and the substitution list are all empty.
3. For an existential quantification, the algorithm tries to go through all the existentially bound variables and tries to eliminate them. To do this the coalesce step is performed for all the existentially bound variables as shown below:

$$E_1\ \exists x\ .(x \equiv t\ ,\ E_2) \quad \rightarrow \quad E_1\ E_2[x/t] \tag{4.1}$$

$$E_1\ \exists x\ .E_2 \quad \rightarrow \quad \exists x\ .E_1\ E_2 \quad x \equiv t \notin E_2 \tag{4.2}$$

   (a) One locates a substitution of an existentially bound variable and uses it to eliminate that variable from the list shown above in (4.1). This is done by substituting it into the terms of all equations which then must be simplified by matching and occurs checking.

(b) If a substitution doesn't exist for the existential variable, then the type equation on the immediately upper level in pulled inside the existential equation. This is shown above in (4.2).

(c) One keeps doing (a) until there are no more assignments associated with the existentially bound variables. The new free variables are the old free variables less all the variables in the new binding.

4. The algorithm tries to eliminate all the universally bound variables. If the variable x is not present in E then x is removed. If there exists a non trivial substitution for x, then an error is reported.

$$\forall x.E \rightarrow \begin{cases} E & x \notin E \\ \text{error} & \text{Otherwise} \end{cases}$$

5. To add a type equation E_1 to an already processed type equation list E_2(which either fails or returns a package), one evalautes E_1 (by matching) to a package as well. The two packages can then be combined to get a resultant package as shown below:

$$E_1\ E_2 \rightarrow E_1\ (f_2, u_2, e_2, s_2) \rightarrow (f_1, u_1, e_1, s_1)\ (f_2, u_2, e_2, s_2) \rightarrow$$
$$(f_1 +\!\!+ f_2,\ u_1 +\!\!+ u_2,\ e_1 +\!\!+ e_2, s_1 +\!\!+ s_2)$$

The elements in the final package are obtained by concatenating the corresponding elements of the two packages.

6. The final result of this may have a substitution list which still contains repeated assignments to variables (the free ones) and so it must be substituted or matched with respect to the free variables and then substituted out to get the final substitution of the free variables. The final package will not have any universally bound variables but may

still have some existentially bound variables left and the solution is now parametric in these existentially bound variables.

Example

Consider a type equation shown below:

$$\exists X_1, X_2. \left(\begin{array}{l} X_0 \equiv \texttt{List}(X_1), X_1 \equiv X_2, \\ \exists X_3, X_4. X_3 \equiv \texttt{List}(X_4), X_2 \equiv X_3 \end{array} \right)$$

where X_0 is the type variable corresponding to the construct being type inferred.

1. Look at the inner most level and create a package. Create packages corresponding to the equation $X_3 \equiv \texttt{List}\ (X_4)$ and $X_2 \equiv X_3$ and combine them.

   Package for the equation $X_3 \equiv \texttt{List}\ (X_4)$ is $([X_3, X_4], [], [], [X_3 \equiv \texttt{List}\ (X_4)])$

   Package for the equation $X_2 \equiv X_3$ is $([X_2, X_3], [], [], [X_2 \equiv X_3])$

   Combined package is $([X_2, X_3, X_4], [], [], [X_3 \equiv \texttt{List}\ (X_4), X_2 \equiv X_3])$

2. For the inner existentially qunatified equation containing the existential type variables X_3 and X_4, try eliminating the existential type variables using the combined package from the previous step. The type variable X_3 can be removed by substitution but type variable X_4 can not be removed. The package generated is $[X_2], [], [X_4], [X_2 \equiv \texttt{List}(X_4)]$.

3. The package generated in the previous step is combined with the package generated for the type equations $X_0 \equiv \texttt{List}(X_1)$ and $X_1 \equiv X_2$).

   Package generated for the equation $X_0 \equiv \texttt{List}(X_1))$ is $([X_0, X_1], [], [], [X_0 \equiv \texttt{List}(X_1)])$

Package generated for the equation $X_1 \equiv X_2$ is:

$$([X_1, X_2], [], [], [X_1 \equiv X_2])$$

The combined package of the three packages is:

$$\left([X_0, X_1, X_2], [], [], \begin{bmatrix} X_2 \equiv \texttt{List}(4), \\ X_0 \equiv \texttt{List}(1), \\ X_1 \equiv X_2 \end{bmatrix}\right)$$

4. Try eliminating the existentially quantified variable of the outer existentially quantified equation using the combined package obtained in the previous step. The package obtained is $([X_4], [], [], [X_0 \equiv \texttt{List}(\texttt{List}(X_4))])$. The type of the construct is given by:

$$\forall X_4.\texttt{List}(\texttt{List}(X_4))$$

4.2.1 Helper Functions Used in Solving Type Equations

The helper functions used in the type equation solution algorithm are check, match, and coalesce. These functions are shown in Figure 4.3 and described below:

The check function peforms the occurs check. The function takes as input a pair, represented by T/X, consisting of a type expression T and a type variable X. If the type variable X is present in the type expression T then the a singleton list is returned with the pair T/X otherwise *fails occurs check* error is reported. In the special case when the input pair is X/X, an empty list is returned.

The match function produces a list of substitutions by matching the two type expressions of its input pair. A description of the functions is:

$$
\begin{array}{lcl}
\mathsf{check}(X/X) & = & [] \\
\mathsf{check}(T/X) & = & \begin{cases} [T/X] & \text{if } X \text{ does not occur in } T \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
\\
\mathsf{match}(X,T) & = & \mathsf{check}(T/X) \\
\mathsf{match}(T,X) & = & \mathsf{check}(T/X) \\
\mathsf{match}(F(\vec{T}),G(\vec{S})) & = & \begin{cases} \mathsf{flatten}(\mathsf{map\ match\ (zip\ }(\vec{T},\vec{S}))) & \text{if } F \equiv G \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
\\
\mathsf{coalesce}(T/X,[]) & = & [] \\
\mathsf{coalesce}(T/X,S/Y{:}\mathsf{Rest}) & = & \begin{cases} \mathsf{append}(\mathsf{match}(T,S),\mathsf{coalesce}(T/A,\mathsf{Rest})) & \text{if } X \equiv Y \\ \mathsf{append}(\mathsf{check}(S[T/X]/Y),\mathsf{coalesce}(T/A,\mathsf{Rest})) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4.3: Helper Functions Used in the Solution of Type Equations

- If one of the type expression of the input pair is a type variable, then occurs check is performed for the pair getting a substitution.

- If the two type expressions of the input pair are made up of type constructors, say F and G which can be data types, codata types, protocols, coprotocols, functions, or processes, then these steps are performed: Check if the type constructors are the same. If the type constructors are not same, then an error is reported. If the type constructors are the same, then the match functions are applied to the corresponding elements of F and G resulting in a list of list of substitutions which are then concatenated.

The coalesce function takes as input a substitution, and list of substitutions. For a substitution T/X, the function goes through the list of substituions. If the head of the substitution list is a substitution of the variable X, the term T and the term corresponding to the new substitution of X are matched to get a list of susbtitutions which are then appended to the output obtained by applying the coalesce function on T/X and the tail of the substitution list. If the head of the substitution list of the coalesce function contains the variable X in the term, then T can be substituted for X in the term.

4.3 Role of Symbol Table in Type Inference

Symbol table acts a repository of symbols during the type inference process. The types of the various constituents of an MPL program, constructors of data definitions, destructors of codata definitions, (co)handle of a (co)protocol definitions, function definitions, and process definitions, are inserted in the symbol table. When these constituents are used later in the program, their types can be looked up from the symbol table in order to generate type equations. For example, consider the `List` data type defined below:

```
data List(A) -> C = Nil  :: -> C
                  Cons :: A,C -> C
```

In the symbol table, the constructors `Nil`, and `Cons` are inserted along with their fold types, constructor types, and the data type they belong to. So, for instance, if one is trying to type infer the term `Cons(1,Nil)` then type of `1` which is `Int` is unified with `A` generating the type equation `A = Int`. Once can then replace the variable `A` with `Int` in the data type of the constructor, `List(A)`, producing `List(Int)`, which is the inferred type of the aforementioned constructor term.

Chapter 5

Type Equations for Sequential MPL

In this chapter, the rules for generating type equations for the sequential MPL constructs are described. The type equation generation rules are term formation rules annotated with the typing information for terms.

5.1 Data Type Constructs

The case, the constructor, and the fold are the sequential MPL constructs that work on the data types. This section describes a data type declaration in MPL followed by a description of the type equations for the various data type constructs.

5.1.1 Populating the Symbol Table for Data Declaration

Consider a pair of mutually recursive data types: $D_1(A_1, \ldots, A_k)$ and $D_2(A_1, \ldots, A_k)$ defined below:

data

$$D_1(A_1, \ldots, A_k) \;\rightarrow\; Z_1 =$$

$$C_{11} : T_{11,1}, \ldots, T_{11,n_{11}} \quad \rightarrow \quad Z_1$$

$$
\begin{array}{lll}
\vdots & \vdots & \vdots \\
C_{1m} : T_{1m,1}, \ldots, T_{1m,n_{1m}} & \rightarrow & Z_1
\end{array}
$$

and

$$
\begin{array}{l}
D_2(A_1, \ldots, A_k) \ \rightarrow \ Z_2 = \\
\qquad \begin{array}{lll}
C_{21} : T_{21,1}, \ldots, T_{21,n_{21}} & \rightarrow & Z_2 \\
\vdots & \vdots & \vdots \\
C_{2r} : T_{2r,1}, \ldots, T_{2r,n_{2r}} & \rightarrow & Z_2
\end{array}
\end{array}
$$

In the data type definitions D_1 and D_2, the variables in which the data types are polymorphic are the same. In general this might not to be true and the "recursion graph" between the various data types, defined with the syntax of mutually recursive data type definitions, must be constructed in order to find out the variable containment for each data type.

The properties of a data type definition have been discussed in Section 2.1. Here we only focus on those aspects of a data type definition which are important for type inference. For the i^{th}constructor C_{1i} of the data type D_1:

- The type of the phrase of a fold function corresponding to the constructor is given by the type expression for that constructor in the data definition. The type of the fold for constructor C_{1i}, held in the symbol table, is:

$$\forall \ Z_1, A_1, \ldots, A_k \ . \ T_{i1,1}, \ldots, T_{i1,n_{i1}} \rightarrow Z_1 \tag{5.1}$$

  where $Z_1, A_1, \ldots, A_k$ represents all the type variables present in the data definition D_1.

  The type of the fold phrase stored in the symbol table is used in the generation of type equations for the fold construct. When the type equations are generated, the type of

the fold phrase is α-renamed to avoid any naming conflicts with the variables already present in the type equation. The type of the fold phrase for the constructor C_{1i} with fresh variables $Z'_1, A'_1, \ldots, A'_k$ substituted for $Z_1, A_1, \ldots, A_k$ is:

$$(\forall\ Z_1, A_1, \ldots, A_k\ .\ T_{i1,1}, \ldots, T_{i1,n_{i1}} \rightarrow Z_1)\ \ Z'_1\ A'_1 \ldots\ A'_k$$

which can be β-reduced to:

$$(T_{1i,1}, \ldots, T_{1i,n_{1i}} \rightarrow Z_1)\ \ [Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k].$$

The above notation is used in the type equations for the various data type constructs.

- The type of the constructor C_{1i} is given by substituting the state variable Z_1 with the data name $D_1(A'_1, \ldots, A'_k)$, where $A'_1, \ldots, A'_k$ are fresh variables used for $A_1, \ldots, A_k$ respectively, in the body of the type expression for the constructor in the data definition. The type of C_{1i}, held in the symbol table, is:

$$(\forall\ A'_1, \ldots, A'_K\ (\forall\ Z_1, A_1, \ldots, A_K\ .\ T_{1i,1}, \ldots, T_{1i,n_{i1}}\ \rightarrow\ Z_1)\ D_1(A'_1, \ldots, A'_k))\ A'_1 \ldots A'_K$$

which can be β-reduced to

$$(T_{1i,1}, \ldots, T_{1i,n_{1i}}\ \rightarrow\ Z_1)[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k]$$

5.1.2 The Constructor

A data type has a set of constructors. The constructors of a data type are used to generate instances of that data type. Table 5.1 shows the type equations for the constructors. The constructor C_{1i}, for $1 \leq i \leq m$, used in the Table 5.1 is a constructor of the data type $D_1(A_1, \ldots, A_k)$ defined in Section 5.1.1, takes j arguments. A description of the type

$$\frac{\Gamma \vdash t_1 : T_1 \quad \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash t_{n_{1i}} : T_(n_{1i}) \quad \langle E_{n_{1i}} \rangle}{\Gamma \vdash \mathsf{cons}\ (C_{1i}, (t_1, \ldots, t_{n_{1i}}))\ :\ T}\ \text{cons}$$

$$\left\langle \exists\ \begin{matrix} T_1, \ldots, T_{n_{1i}}, \\ A'_1, \ldots, A'_k. \end{matrix} \begin{matrix} T = D_1(A'_1, \ldots, A'_k), \\ T_1 = T_{1i,1}\ \Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \vdots \\ T_{n_{1i}} = T_{1i,n_{1i}}\ \Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ E_1, E_2, \ldots, E_{n_{1i}} \end{matrix} \right\rangle$$

Table 5.1: Type Equations for Constructors

equations for the constructor is:

- The output type T of the constructor C_{i1} is the data type of which it is a constructor. The equation generated is:

$$T = D_1(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables.

- The type of the k^{th} argument t_k of the constructor C_{1i} is equated with the type of the corresponding element of the input type of constructor function of C_i obtained from the symbol table. The equation generated is:

$$T_k = T_{1i,k}\Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \quad \{1 \leq k \leq n_{1i}\}$$

  where T_k is the type of t_k, and $T_{1i,k}\Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big]$ represents the α-renamed k^{th} element of the input type of the constructor C_{1i}.

- E_1, ..., $E_{n_{1i}}$ represent the set of type equations generated for the terms t_1, ..., $t_{n_{1i}}$ respectively.

$$\frac{\Gamma \vdash t : T_0 \ \langle E_0 \rangle \qquad \begin{array}{lll} \Gamma, x_{11,1} : F_{11,1}, \ldots, x_{1m,n_{1m}} : F_{1m,n_{1m}} & \vdash & t_1 : T_1 \quad \langle E_1 \rangle, \\ \vdots & & \\ \Gamma, x_{1m,1} : F_{1m,1}, \ldots, x_{1m,n_{1m}} : F_{1m,n_{1m}} & \vdash & t_m : T_m \quad \langle E_m \rangle \end{array}}{\Gamma \vdash \begin{array}{ll} \mathsf{case} & t \\ \mathsf{of} & \left| \begin{array}{lll} C_{11}\ x_{11,1}, \ldots, x_{11,n_{11}} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ C_{1m}\ x_{1m,1}, \ldots, x_{1m,n_{1m}} & \rightarrow & t_m \end{array} \right. \end{array} : T} \ \text{case}$$

$$\left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_m, \\ F_{11,1}, \ldots, F_{11,n_{11}}, \\ \quad \vdots \qquad . \\ F_{1m,1}, \ldots, F_{1m,n_{1m}}, \\ A'_1, \ldots, A'_k \end{array} \begin{array}{l} T_0 = D_1(A'_1, \ldots, A'_k), \\ T_1 = T, \ldots, T_m = T \\ F_{11,1} = T_{11,1}\Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \quad \vdots \\ F_{11,n_{11}} = T_{11,n_{11}}\Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \quad \vdots \\ F_{1m,1} = T_{1m,1}\Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \quad \vdots \\ F_{1m,n_{1m}} = T_{1m,n_{1m}}\Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ E_0, E_1, E_2, \ldots, E_m \end{array} \right\rangle$$

Table 5.2: Type Equations for case

5.1.3 The case

The case construct branches on the different constructors of a data type. Table 5.2 shows the type equations for a case construct over the data type D_1, which was defined in Section 5.1.1. A description of these equations is:

- The type T_0 of the term t_0, which is the term being cased on, should be a data type D_1 of which $C_{11}, \ldots, C_{1m}$ are constructors. The data type of these constructors is looked up from the symbol table. The type equation generated is:

$$T_0 = D_1(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables.

- E_0 represents the set of type equations generated for the term t.

- The output type T of the case term is the type of the terms $t_1, \ldots, t_m$. Thus, in a well formed case expression the types of all the terms $t_1, \ldots, t_m$ should be the same. The type equation generated is:

$$T_1 = T, \ldots, T_m = T$$

- The type $F_{1i,j}$ of the term $x_{1i,j}$, which is the j^{th} argument of the constructor C_{1i}, is equated with the type of the j^{th} element of the input type of the constructor C_i, obtained from the symbol table. The type equation generated is:

$$F_{1i,j} = T_{1i,j}\Big[D_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \quad \{1 \leq i \leq m, 1 \leq j \leq n_j\}$$

- E_i, for $1 \leq i \leq m$, is the set of type equations generated for the term t_i, which is the term associated with the constructor C_{1i} in the case branch, in the context expanded by the n_{1i} variables, $x_{1i,1}, \ldots, x_{1i,n_{i1}}$ of types $F_{1i,1}, \ldots, F_{1i,n_{1i}}$ respectively.

5.1.4 The fold

The fold construct is used to implement recursion in a disciplined manner. Recall, from the first chapter on sequential MPL, that folding over mutually recursive data types requires the constructors of each reachable mutually recursive data type to be listed.

The fold construct over the data type D_1, which is mutually recursive with the data type D_2 (defined in Section 5.1.1), requires the constructors of both the data types D_1 and D_2 to be listed. Table 5.3 describes the type equations for a fold construct for the D_1 mutually recursive with D_2. A description of the type equations is:

- The type T_0 of the term t which is the subject of the fold should be a data type. This should be the same data type to which the first constructors $C_1, \ldots, C_m$ belongs. The

$$\frac{\begin{array}{lll} x_{11,1} : F_{11,1}, \ldots, x_{11,n_{11}} : F_{11,n_{11}} & \vdash \; t_1 : T_1 & \langle E_1 \rangle \\ \vdots & \vdots \quad \vdots & \\ x_{1m,1} : F_{1m,1}, \ldots, x_{1m,n_{1m}} : F_{1m,n_{1m}} & \vdash \; t_m : T_m & \langle E_m \rangle \\ \\ y_{21,1} : G_{21,1}, \ldots, y_{21,n_{21}} : G_{21,n_{21}} & \vdash \; u_1 : U_1 & \langle E'_1 \rangle \\ \vdots & \vdots \quad \vdots & \\ y_{2r,1} : G_{2r,1}, \ldots, y_{2r,n_{2r}} : G_{2r,n_{2r}} & \vdash \; u_r : U_r & \langle E'_r \rangle \end{array}}{\begin{array}{l} \Gamma \vdash \; \mathsf{fold} \;\; t \\ \qquad \mathsf{of} \;\; \left| \begin{array}{lll} C_{11} \; : \; x_{11,1}, \ldots, x_{11,n_{11}} & \to \; t_1 & \\ \vdots & \vdots \quad \vdots & \\ C_{1m} \; : \; x_{1m,1}, \ldots, x_{1m,n_{1m}} & \to \; t_m & : T \\ C_{21} \; : \; y_{21,1}, \ldots, y_{21,n_{21}} & \to \; u_1 & \\ \vdots & \vdots \quad \vdots & \\ C_{2r} \; : \; y_{2r,1}, \ldots, y_{2r,n_{2r}} & \to \; u_r & \end{array} \right. \end{array}} \text{fold}$$

$t : T_0 \; \langle E_0 \rangle$

$$\left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_m, \\ F_{11,1}, \ldots, F_{11,n_{11}}, \\ \quad \vdots \qquad . \\ F_{1m,1}, \ldots, F_{1m,n_{1m}}, \\ G_{21,1}, \ldots, G_{21,n_{21}}, \\ \quad \vdots \qquad . \\ G_{2r,1}, \ldots, G_{2r,n_{2r}}, \\ A'_1, \ldots, A'_k \end{array} \right.$$

$$\begin{array}{l} T_0 = D_1(A'_1, \ldots, A'_k), \\ T_1 = T, \ldots, T_m = T \\ F_{11,1} = T_{11,1} \left[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ F_{11,n_{11}} = T_{11,n_{11}} \left[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ F_{1m,1} = T_{1m,1} \left[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ F_{1m,n_{1m}} = T_{1m,n_{1m}} \left[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k \right], \\ G_{21,1} = T_{21,1} \left[Z'_2/Z_2, A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ G_{21,n_{21}} = T_{21,n_{21}} \left[Z'_2/Z_2, A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ G_{2r,1} = T_{2r,1} \left[Z'_2/Z_2, A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ G_{2r,n_{2r}} = T_{2r,n_{2r}} \left[Z'_2/Z_2, A'_1/A_1, \ldots, A'_k/A_k \right], \\ E_0, E_1, \ldots, E_m, \\ E'_1, \ldots, E'_r \end{array} \right\rangle$$

Table 5.3: Type Equations for fold

data type of these constructors is obtained by looking in the symbol table. The type equation generated is:

$$T_0 = D_1(A'_1, \ldots, A'_k)$$

where $A'_1, \ldots, A'_k$ are fresh variables.

- E_0 represents the set of type equations generated for the term t.

- The output type T of the fold construct is the type of the terms t_1. Thus, in a well formed fold expression the types of $t_1, \ldots, t_m$ should be the same. The type equations generated are:

$$T_1 = T, \ldots, T_m = T$$

where $T_1, \ldots, T_m$ are the types of $t_1, \ldots, t_m$ respectively.

- The type $F_{1i,j}$ of the term $x_{1i,j}$, which is the j^{th} argument of the constructor C_{1i}, is equated with the type of the j^{th} element of the input type of the fold phrase of the constructor C_{1i} obtained from the symbol table. The type equation generated is:

$$F_{1i,j} = T_{1i,j}\Big[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \quad \{1 \leq i \leq m \ , \ 1 \leq j \leq n_{1i}\}$$

- The type $G_{2p,q}$ of the term $y_{2p,q}$, which is the q^{th} argument of the constructor C_{2p}, is equated with the type of the q^{th} element of the input type of the fold phrase of the constructor C_{2p} obtained from the symbol table. The type equation generated is:

$$G_{2p,q} = T_{2p,q}\Big[Z'_2/Z_2, A'_1/A_1, \ldots, A'_k/A_k\Big] \quad \{1 \leq p \leq r, \ 1 \leq q \leq n_{2p}\}$$

- E_i, for $1 \leq i \leq m$, represents the set of type equations generated for the term t_i

in the context expanded by the n_{1i} variables, $x_{1i,1}, \ldots, x_{1i,n_{1i}}$ of types $F_{1i,1}, \ldots, F_{1i,n_{1i}}$ respectively, associated with the constructor C_i in the body of the fold construct.

- E'_p, for $1 \leq p \leq r$, represents the set of type equations generated for the term u_p in the context expanded by the n_{2p} variables, $y_{2p,1}, \ldots, y_{2p,n_{2p}}$ of types $G_{2p,1}, \ldots, G_{2p,n_{2p}}$ respectively, associated with the constructor C_{2p} in the body of the fold construct.

5.2 Codata Constructs

The record, the product, the destructor, and the unfold are the sequential MPL constructs that work on codata types. In this section, the type equations for these constructs are discussed.

5.2.1 Populating the Symbol Table for Codata Declaration

Consider a pair of mutually recursive codata type definitions C_1 and C_2 having destructors $D_1, \ldots, D_m$ and $Q_1, \ldots, Q_r$ respectively:

```
codata
```

$$
\begin{aligned}
&Z_1 \ \rightarrow \ C_1(A_1, \ldots, A_k) = \\
&\quad D_{11} : Z_1,\ T_{11,1}, \ldots,\ \ T_{11,n_{11}} \quad \rightarrow \ P_1 \\
&\quad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
&\quad D_{1m} : Z_1,\ T_{1m,1}, \ldots, T_{1m,n_{1m}}, \quad \rightarrow \ P_m
\end{aligned}
$$

```
and
```

$$
\begin{aligned}
&Z_2 \ \rightarrow \ C_2(A_1, \ldots, A_k) = \\
&\quad D_{21} : Z_2,\ T_{21,1}, \ldots, T_{21,n_{21}} \quad \rightarrow \ S_1 \\
&\quad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
&\quad D_{2r} : Z_2,\ T_{2r,1}, \ldots, T_{2r,n_{2r}} \quad \rightarrow \ S_r
\end{aligned}
$$

For the codata type C_1:

- The type of the phrase of a unfold function for the destructor D_{1i} is given by the type expression corresponding to the destructor in the codata definition. The type of the unfold for the destructor D_{1i}, which is held in the symbol table, is:

$$\forall\ Z_1, A_1, \ldots, A_k.Z_1, T_{1i,1}, \ldots, T_{1i,n_{1i}} \rightarrow P_i \qquad \{1 \leq i \leq m\} \tag{5.2}$$

  where $A_1, \ldots, A_k$ represents the set of the type variables present in the type expression.

  The type of the unfold phrase stored in the symbol table is used in the generation of type equations for the unfold construct. The unfold type of the destructors are α-renamed with fresh variables to avoid variable name collision in a type equation. The α-renamed unfold type of the destructor D_{1i} with fresh variables $Z'_1, A'_1, \ldots, A'_k$ substituted for $Z_1, A_1, \ldots, A_k$ is:

$$(\forall Z_1, A_1, \ldots, A_k.Z_1, T_{1i,1}, \ldots, T_{1i,n_{1i}}\ \rightarrow\ P_i)\ Z'_1, A'_1, \ldots, A'_k$$

  which can be β-reduced to:

$$(Z_1, T_{1i,1}, \ldots, T_{1i,n_{1i}}\ \rightarrow\ P_i)\ [Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k]$$

- The type of the destructor D_{1i} is given by substituting the state variable Z_1 with the codata name $C_1(A'_1, \ldots, A'_k)$, where $A'_1, \ldots, A'_k$ are fresh variables used for $A_1, \ldots, A_k$ respectively, in the type expression for the destructor given in the codata definition. The type of the destructor D_{1i}, held in the symbol table, is:

$$(\forall\ A'_1, \ldots, A'_k(\forall\ Z_1, A_1, \ldots, A_k\ .\ Z_1, T_{i1}, \ldots, T_{ij} \rightarrow P_i)\ C_1(A'_1, \ldots, A'_k))\ A'_1 \ldots A'_k$$

$$\frac{\begin{array}{lll} x_{11,1} : F_{11,1}, \ldots, x_{11,n_{11}} : F_{11,n_{11}} & \vdash & t_1 : T_1 \quad \langle E_1 \rangle \\ \vdots & & \vdots \\ x_{1m,1} : F_{1m,1}, \ldots, x_{1m,n_{1m}} : F_{1m,n_{1m}} & \vdash & t_m : T_m \quad \langle E_m \rangle \end{array}}{\begin{array}{l} \Gamma \vdash \mathsf{rec} \\ \quad \mathsf{of} \\ \left| \begin{array}{lll} D_{11} \ : \ x_{11,1}, \ldots, x_{11,n_{11}} & \to & t_1 \\ \vdots & & \vdots \\ D_{1m} \ : \ x_{1m,1}, \ldots, x_{1m,n_{1m}} & \to & t_m \end{array} \right. \end{array} : T} \ \text{rec}$$

$$\left\langle \exists \begin{array}{l} T_1, \ldots, T_m, \\ F_{11,1}, \ldots, F_{11,n_{11}}, \\ \quad \vdots \qquad . \\ F_{1m,1}, \ldots, F_{1m,n_{1m}}, \\ A'_1, \ldots, A'_k \end{array} \begin{array}{l} T = C_1(A'_1, \ldots, A'_k) \\ T_1 = P_1\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \qquad \vdots \\ T_m = P_m\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ F_{11,1} = T_{11,1}\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \\ \qquad \vdots \\ F_{11,n_{11}} = T_{11,n_{11}}\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \\ \qquad \vdots \\ F_{1m,1} = T_{1m,1}\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \\ \qquad \vdots \\ F_{1m,n_{1m}} = T_{1m,n_{1m}}\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \\ E_1, E_2, \ldots, E_m \end{array} \right\rangle$$

Table 5.4: Type Equations for record

which can be β-reduced to:

$$(Z_1, T_{i1}, \ldots, T_{ij} \to P_i) \ \ [C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots A'_k/A_k]$$

5.2.2 The record

The record construct forms a record of a codata type. The products allow the programmers to use products in MPL without specifying the corresponding codata definitions with the

various projections as destructors. A description of the type equations generated for the record construct, shown in the first row of the Table 5.4, is:

- The type T of the record is the codata type of the destructors $D_1, \ldots, D_m$ used in the body of the record construct. The type equation generated is:

$$T \;=\; C_1(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables.

- The type T_{1i} of the term t_i, which is the term corresponding to the destructor D_{1i} in the record, is equated with the output type of destructor function type for D_{1i} obtained from the symbol table. The type equation generated is:

$$T_i = P_i\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \quad \{1 \leq i \leq m\}$$

- The type $F_{1i,j}$ of $x_{1i,j}$, which is the j^{th} argument of the i^{th} destructor D_{1i} of the codata type C_1, is given by the equation:

$$F_{1i,j} = T_{1i,j}\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big] \quad \{1 \leq i \leq m \;, 1 \leq j \leq n_{1i}\}$$

- E_i, for $1 \leq i \leq m$, is the set of the type equations generated for the term t_i. E_i is generated in a context formed by the variables $x_{1i,1}, \ldots, x_{1i,n_{1i}}$ with their types $F_{1i,1}, \ldots, F_{1i,n_{1i}}$ respectively.

5.2.3 The Product

Table 5.5 shows the type equations for the products. A description of these equations is:

- The type T of the product of n terms, $t_1, \ldots, t_n$, is a product type comprised of the

$$\frac{\Gamma \vdash t_1 : T_1 \quad \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash t_n : T_n \quad \langle E_n \rangle}{\Gamma \vdash (t_1, \ldots, t_n) : T \quad \left\langle \exists\, T_1, \ldots, T_n.\ \begin{array}{l} T = (T_1, \ldots, T_n) \\ , E_1, \ldots, E_n \end{array} \right\rangle}\ \text{prod}$$

Table 5.5: Type Equations for Products

$$\frac{\begin{array}{l} \Gamma \vdash a_1 : T_1 \quad \langle E_1 \rangle \\ \qquad \vdots \\ \Gamma \vdash a_j : T_j \quad \langle E_j \rangle \qquad r : R \quad \langle E_r \rangle \end{array}}{\Gamma \vdash \mathsf{dest}\ (D_i, [r, a_1, \ldots, a_j]) : T}\ \text{dest}$$

$$\left\langle \exists\ \begin{array}{l} T_0, T_1, \ldots, T_j, \\ A'_1, \ldots, A'_k \end{array}\ .\ \begin{array}{l} R = C_1(A'_1, \ldots, A'_k), \\ T = P_i\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ T_1 = T_{1i,1}\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \qquad \vdots \\ T_j = T_{1i,n_{1i}}\Big[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ , E_1, \ldots, E_j, E_r \end{array} \right\rangle$$

Table 5.6: Type Equations for Destructor

types of the individual terms. The type equation generated is:

$$T = (T_1, \ldots, T_n)$$

where $T_1, \ldots, T_n$ represents the types of the terms $t_1, \ldots, t_n$ respectively.

- $E_1, \ldots, E_n$ represent the set of type equations generated for the terms $t_1, \ldots, t_n$ respectively.

5.2.4 The Destructor

One can destruct a record of a codata type and extract a value using the destructors of that codata type. A description of the type equations for a destructor, shown in Table 5.6, is:

- The destructor D_{1i} of the codata type C_1 takes j arguments, $a_1, \ldots, a_j$, in addition to the first term r which is the value being destructed. The type R of the term r is the codata type to which the destructor D_i belongs. The type equation generated is:

$$R = C_1(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables.

- The output type T of the destructor term D_{1i} is equated with the output type of the destructor D_{1i}, obtained from the symbol table. The type equation generated is:

$$T = P_i[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k]$$

- The type T_k of the argument a_k, which is the $(k+1)^{th}$ input to the destructor D_{1i}, is equated with the corresponding input type of the destructor D_{1i}. The type equation generated is:

$$T_k = T_{1i,k+1}[C_1(A'_1, \ldots, A'_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k] \quad \{1 \leq k \leq j\}$$

- $E_1, \ldots, E_j$ represent the set of equations generated for the arguments $a_1, \ldots, a_j$ respectively.

5.2.5 The unfold

The unfold is a operation used to produce codata by unfolding over an initial value.

Suppose, the codata defined in Section 5.2.1 is not a mutually recursive data type. The type equations for the unfold construct over the codata C_1 is shown in Table 5.7. A description of type equations for the unfold construct is:

- The output type T of an unfold construct is the type T_0 of the term t which is being

$$
\begin{array}{llll}
x_{11,1} : F_{11,1}, \ldots, x_{11,n_{11}} : F_{11,n_{11}}, s : S_1 & \vdash & t_1 : T_1 & \langle E_1 \rangle, \\
\vdots & \vdots & \vdots & \\
x_{1m,1} : F_{1m,1}, \ldots, x_{1m,n_{1m}} : F_{1m,n_{1m}}, s : S_m & \vdash & t_m : T_m, & \langle E_m \rangle
\end{array}
$$

$t : T_0 \ \langle E_0 \rangle$

$$
\Gamma \vdash \mathsf{s} \Rightarrow \left| \begin{array}{lllll}
D_{11} & : & x_{11,1}, \ldots, x_{11,n_{11}} & \rightarrow & t_1 \\
\vdots & & \vdots & \vdots & \vdots \\
D_{1m} & : & x_{1m,1}, \ldots, x_{1m,n_{1m}} & \rightarrow & t_m
\end{array} \right. : T
$$

$$
\left\langle \exists \begin{array}{ll}
T_0, T_1, \ldots, T_m, & T = T_0, \\
F_{11,1}, \ldots, F_{11,n_{11}}, & T_0 = C_1(A'_1, \ldots, A'_k), \\
\quad \vdots \quad . & S_1 = T_0, \ldots, S_m = T_0, \\
F_{1m,1}, \ldots, F_{1m,n_{1m}}, & T_1 = P_1\Big[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\
A'_1, \ldots, A'_k, & \qquad \vdots \\
& T_m = P_m\Big[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\
& F_{11,1} = T_{11,1}\Big[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\
& \qquad \vdots \\
& F_{11,n_{11}} = T_{11,n_{11}}\Big[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\
& \qquad \vdots \\
& F_{1m,1} = T_{1m,1}\Big[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\
& \qquad \vdots \\
& F_{1m,n_{1m}} = T_{1m,n_{1m}}\Big[Z'_1/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\
& E_0, E_1, \ldots, E_m
\end{array} \right\rangle
$$

Table 5.7: Type Equations for unfold

unfolded. The type equation generated is:

$$T = T_0$$

- The types of the term t is the codata type to which the destructors belong to. The type equations generated are:

$$T_0 = C_1(A'_1, \ldots, A'_k)$$

where T_0 is the type of t, and $A'_1, \ldots, A'_k$ are fresh variables.

- E_0 is the set of type equations generated for the term t.

- E_i, for $1 \leq i \leq m$, represents the set of type equations generated for the term t_i. E_i is generated in the context expanded by the n_{1i} variables, $x_{1i,1}, \ldots, x_{1i,n_{1i}}$ of types $F_{i1,1}, \ldots, F_{1i,n_{1i}}$ respectively, associated with the destructor D_i in addition to the state variable s of type S_i. The variables s represents the state for the codata types C_1.

- The type equations generated for the types associated with the state s are:

$$S_1 = T_0, \ldots, S_m = T_0,$$

where $S_1, \ldots, S_m$ are the types of the state s_1 in the unfold branch corresponding to the terms $t_1, \ldots, t_m$ respectively.

- The type T_i of term t_i is equated with the output type of the unfold function for the

destructor D_i, obtained from the symbol table. The type equation generated is:

$$T_i = P_i\Big[Z_1'/Z_1, A_1'/A_1, \ldots, A_k'/A_k\Big] \quad \{1 \leq i \leq m\}$$

- The type $F_{1i,j}$ of $x_{1i,j}$, which is the j^{th} argument of the i^{th} destructor (D_{1i}) phrase, is equated with the corresponding input type of the unfold function of the destructor D_{1i} looked up from the symbol table. The type equation generated is:

$$F_{1i,j} = T_{1i,j}\Big[Z_1'/Z_1, A_1'/A_1, \ldots, A_k'/A_k\Big] \quad \{1 \leq i \leq m \ , \ 1 \leq j \leq n_{1i}\}$$

5.2.6 Other Sequential MPL Constructs

These are the remaining sequential MPL constructs which are neither the the data or the codata constructs. These include variables, constants, function call, if-then-else, switch and where. This section describes the type equations for these constructs.

5.2.7 Variables

The first rule in the Table 5.8 shows the type equation for variables. The variable x for which the type equation is being generated should be present in the context Γ otherwise an error is reported. The type T of the variable x is equated with the type P, which is the type of the variable x looked up from the context. The type equation generated is:

$$T = P$$

5.2.8 Constants

The type equations for the various kinds of constants are the second through the fourth rules in the Table 5.8. The constants are particularly easy to type infer as different constants are

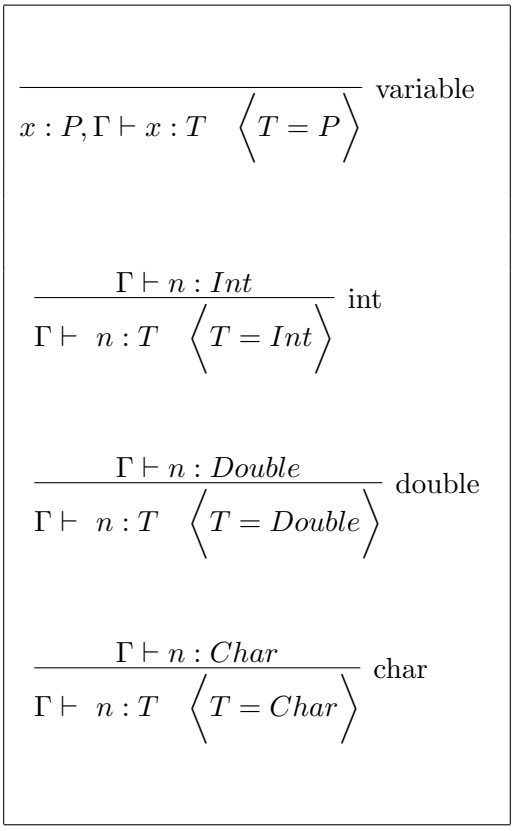

Table 5.8: Type Equations for Variables and Constants

$$\frac{\Gamma \vdash t_1 : T_1 \;\; \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash t_m : T_m \;\; \langle E_m \rangle}{\Gamma \vdash \mathsf{f}\ (\mathsf{t_1}, \ldots, \mathsf{t_m}) : T \;\; \left\langle \exists \begin{array}{l} A'_1, \ldots, A'_k, \\ T_1, \ldots, T_m. \end{array} \begin{array}{l} T = S\ \left[A'_1/A_1, \ldots, A'_k/A_k\right], \\ T_1 = S_1\ \left[A'_1/A_1, \ldots, A'_k/A_k\right], \\ \vdots, \\ T_m = S_m\ \left[A'_1/A_1, \ldots, A'_k/A_k\right], \\ E_1, \ldots, E_m \end{array} \right\rangle} \text{ call}$$

Table 5.9: Type Equations for Function Call

represented with different data constructors in the abstract syntax tree of MPL.

5.2.9 Function Calls

An already defined function can be called with some arguments. Consider a function f which takes m inputs of types $S_1, \ldots, S_m$. The output type of the function is S. The type of the function f, held in the symbol table, is:

$$\mathsf{f} : \forall A_1, \ldots, A_k . S_1, \ldots, S_m \rightarrow S$$

where $A_1, \ldots, A_k$ represents the type variables present in the input and the output types. The type of the function being called is looked up from the symbol table which is then used in the generation of type equations for the function call. The output and input types of the function extracted from the symbol table are α-renamed before using them in the type equation. The α-renamed input and output types are:

$$S_i \left[A'_1/A_1, \ldots, A'_k/A_k\right] \quad \{\ 1 \ \leq \ i \ \leq m\} \qquad (\mathsf{input\ \ type})$$

$$S \left[A'_1/A_1, \ldots, A'_k/A_k\right] \qquad (\mathsf{output\ \ type})$$

where $A'_1, \ldots, A'_k$ are fresh variables and $\left[A'_1/A_1, \ldots, A'_k/A_k\right]$ represents a list of substitutions in the type expressions S_i and S.

A description of the type equations for a function call, shown in Table 5.9, is:

$$\frac{\Gamma \vdash t_1 : T_1 \;\; \langle E_1 \rangle \qquad \Gamma \vdash t_2 : T_2 \;\; \langle E_2 \rangle \qquad \Gamma \vdash t_3 : T_3 \;\; \langle E_3 \rangle}{\Gamma \vdash \textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3 : T \quad \left\langle \exists\, T_1, T_2, T_3. \begin{array}{l} T_1 = Bool, \\ T_2 = T, T_3 = T, \\ E_1, E_2, E_3 \end{array} \right\rangle} \text{ if}$$

$$\frac{\begin{array}{ll} \Gamma \vdash p_1 : P_1 \;\; \langle E_{p,1} \rangle & \Gamma \vdash t_1 : T_1 \;\; \langle E_1 \rangle \\ \vdots & \vdots \\ \Gamma \vdash p_m : P_m \;\; \langle E_{p,m} \rangle & \Gamma \vdash t_m : T_m \;\; \langle E_m \rangle \end{array}}{\Gamma \vdash \begin{array}{l}\textsf{switch} \\ \left|\begin{array}{lcl} p_1 & = & t_1 \\ \vdots & \vdots & \vdots \\ p_m & = & t_m \end{array}\right. \end{array} : T \quad \left\langle \exists \begin{array}{l} T_1, \ldots, T_m, \\ P_1, \ldots, P_m. \end{array} \begin{array}{l} P_1 = Bool, \ldots, P_m = Bool \\ T_1 = T, \ldots, T_m = T, \\ E_1, \ldots, E_m, \\ E_{p,1}, \ldots, E_{p,m} \end{array} \right\rangle} \text{ switch}$$

Table 5.10: Type Equations for if-then-else, and switch

- The type T of a function call for function f is the output type of the function. The type equation generated is:

$$T = S\ \left[A'_1/A_1, \ldots, A'_k/A_k\right]$$

where S is the output type of f and A'_1, A'_k are fresh variables.

- The type S_i of the i^{th} argument of the function call is equated with T_i, which is the type of the i^{th} element of the input type of the function type of f looked up from the symbol table. The type equation generated is:

$$T_i = S_i\ \left[A'_1/A_1, \ldots, A'_k/A_k\right] \quad \{1 \leq i \leq m\}$$

where $A'_1, \ldots, A'_k$ are fresh variables.

5.2.10 The if-then-else

The if-then-else term takes three terms as arguments: the first argument evaluates to a boolean value and of the remaining two arguments, one is associated with the then clause and the other is associated with the else clause. Depending on the boolean value being True or False, first or the second term is executed. A description of the type equations for the if-then-else term, shown as the first rule in Table 5.10, is:

- The type T_1 of the first term t_1 is boolean. The type equation generated is:

$$T_1 = Bool$$

- The type T of the if-then-else term is the type of the terms $t2$ and $t3$, which are the terms corresponding to the then clause and the else clause respectively. Thus, in a well formed if-then-else term, the types of both t_2 and t_3 should be the same. The type equation generated is:

$$T_2 = T \ , \ T_3 = T$$

  where T_2 and T_3 are the types of the terms t_2 and t_3 respectively.

5.2.11 The switch

The switch term, also known as boolean guards, consists of a list of a pair of terms. The first term of each pair evaluates to a boolean value. The second term of the pair, the first term of which evaluates to `True`, is executed. In case the first terms of multiple pairs of terma evaluate to true, the first such pair is selected.

A description of the type equations for switch term, shown as the second rule in the Table 5.10 are:

- The type of each of the first elements of the m pairs is $Bool$. The type equation generated is:

$$P_1 = Bool, \ldots, P_m = Bool.$$

where $P_1, \ldots, P_m$ are the types of the first elements of the m pairs of the switch term respectively.

- The output type T of the switch term is the type of the second elements of the pairs. Thus, in a well formed switch term, the types of all the second elements of the pairs should be the same. The type equation generated is:

$$T_1 = T, \ldots, T_m = T$$

where $T_1, \ldots, T_m$ represent the types of the terms $t_1, \ldots, t_m$ respectively.

- $E_{p,1}, \ldots, E_{p,m}$ represent the set of type equations generated for the first terms of the m pairs.

- $E_1, \ldots, E_m$ represent the set of type equations generated for the second terms of m pairs.

5.2.12 The where Clause

The where clause allows programmers to define local constants and local functions. A description of the type equations for the where clause, which are shown below, is:

$$\frac{\Gamma \vdash t : T_0 \quad \langle E_0 \rangle}{\begin{array}{l} t \\ \mathsf{where} \\ \Gamma \vdash \left| \begin{array}{ll} \mathsf{fdefn}_1, & : T \\ \quad \vdots & \\ \mathsf{fdefn}_n & \end{array} \right. \quad \left\langle \exists\, T_0.\ T = T_0, E_0 \right\rangle \end{array}} \text{ where}$$

- The local function definitions represented by $\mathsf{fdefn}_1, \ldots, \mathsf{fdefn}_n$ in the type equations are type inferred first. The inferred types of these local function definitions are then added to the symbol table. These functions are visible just inside the body of the where term, i.e once outside the where term, the function definitions $\mathsf{fdefn}_1, \ldots, \mathsf{fdefn}_n$ must be removed from the symbol table with their types.

- The output type T of the where term is the type of the term t. The type equation generated is:

$$T = T_0$$

  where T_0 is the type of t.

- E_0 is the set of type equations generated for the term t.

5.3 Type Equations for a Pattern Phrase

A function body is made up of a list of pattern phrases. Recall that a pattern phrase is a list of patterns and the corresponding term in a function definition. The Table 5.11 shows the

type equations for a pattern phrase. The symbol $\models$ represents an empty sequential pattern. A description of the rules is:

- $(p_1, \ldots, p_n \quad \rightarrow \quad t)$ represents a pattern phrase with n patterns $p_1, \ldots, p_n$ and the corresponding term t.

- A Pattern extends the context in addition to generating the type equations. Every pattern p_i generates a new context Γ_i and a set of type equations E_i represented as $\langle \Gamma_i, E_i \rangle$. The pair $\langle \Gamma_i, E_i \rangle$ is generated in an empty context.

- E represents the set of the type equations generated for the term t of the pattern phrase. The type equations are generated in the context formed by the union of the original context Γ with the contexts $\Gamma_1, \ldots, \Gamma_n$ generated by the the patterns $p_1, \ldots, p_n$ respectively.

- if $T_1, \ldots, T_n$ are the types of the patterns $p_1, \ldots, p_n$ respectively and T_0 is the type of the term t, then the type T of the pattern phrase is given by the equation:

$$T = T_1, \ldots, T_n \rightarrow T_0$$

  where the symbol $\rightarrow$ in the type expression shows that the type of a pattern phrase is a function type.

The type equations for a pattern phrase use the type equations of the patterns in the pattern phrase. The type inference rules for various patterns were discussed in the Section 5.4.

5.4 Type Equations for Patterns

MPL provides following kinds of patterns:

- Don't Care Pattern

$$\frac{\begin{array}{l}\models\ p_1 : T_1 \quad \Big\langle \Gamma_1, E_1 \Big\rangle \\ \qquad\vdots \\ \models\ p_n\ : T_n \ \Big\langle \Gamma_n, E_n \Big\rangle \qquad\qquad \Gamma\ \cup\ \Gamma_1\ \cup \ldots \cup\ \Gamma_n\ \vdash\ t\ :\ T_0\ \Big\langle E \Big\rangle\end{array}}{\Gamma\ \vdash\ (\mathsf{p_1}, \ldots, \mathsf{p_n}\ \rightarrow\ \mathsf{t}\ ) : T \quad \left\langle \exists\ \begin{array}{l} T_0, \ldots, T_n. \\ \end{array} \begin{array}{l} T = T_1, \ldots, T_n\ \rightarrow\ T_0, \\ E, E_1, \ldots, E_n \end{array} \right\rangle}\ \text{pattPhr}$$

Table 5.11: Type Equations for Pattern Phrase

$$\frac{}{\models\ \_ : T \quad \Big\langle \{\}, \{\} \Big\rangle}\ \text{don't care}$$

$$\frac{}{\models\ \mathsf{VPatt}\ \ \mathsf{x} : T \quad \Big\langle \{x : T\}, \{\} \Big\rangle}\ \text{variable}$$

$$\frac{\begin{array}{l}\models\ p_1 : T_1 \qquad \Big\langle \Gamma_1, E_1 \Big\rangle \\ \qquad\vdots \\ \models\ p_j : T_j \qquad \Big\langle \Gamma_j, E_j \Big\rangle\end{array}}{\models\ \ \mathsf{CPatt}\ \ (\mathsf{C_i}, (\mathsf{p_1}, \ldots, \mathsf{p_j})) : T}\ \text{constructor}$$

$$\left\langle\ \Gamma_1 \cup \ldots \cup \Gamma_j,\ \left\{ \exists\ \begin{array}{l} A_1, \ldots, A_k. \\ T_1, \ldots, T_j \end{array} \begin{array}{l} T = D(A'_1, \ldots, A'_k), \\ T_1 = T_{i1}\Big[D(A_1, \ldots, A_k/Z)\ ,\ A'_1/A_1, A'_k/A_k\Big], \\ \qqu\vdots \\ T_j = T_{ij}\Big[D(A_1, \ldots, A_k/Z)\ ,\ A'_1/A_1, A'_k/A_k\Big], \\ E_1, \ldots, E_j \end{array} \right\}\ \right\rangle$$

Table 5.12: Type Equations for Don't care, Variable, and Constructor Patterns

- Variable Pattern
- Constructor Pattern
- Record Pattern
- Product Pattern

In the following sections, the type equations for the various patterns are discussed. Table 5.12 and 5.13 list out the type equations for various patterns. The type equation generation process for the various patterns yields a context in addition to the type equations.

5.4.1 Don't Care Patterns

A don't care pattern is used in a pattern phrase when a particular input is not relevant for the term of that pattern phrase. The first rule in the Table 5.12 shows the type equations for the don't care pattern. As the rule suggests, the don't care patterns don't contribute anything to either the context or to the set of type equations.

5.4.2 Variable Patterns

In a pattern phrase, the variables introduced by the variable patterns can be used inside the body of the term of that pattern phrase. As can be seen in the second rule in the Table 5.12, a variable pattern extends the context by adding the variable of the pattern with its type variable in the context. However, a variable pattern doesn't add anything to the set of the type equations.

5.4.3 Constructor Patterns

A constructor patterns can be used to pattern match on the constructors of a data type. The third rule in the Table 5.12 shows the type equations for a constructor pattern. The constructor C_i used in the constructor pattern in the rule belongs to the data type $D_1(A_1, \ldots, A_k)$

defined in the Section 5.1.1. A description of the type equations for the constructor pattern of constructor C_i, which takes j arguments, is:

- $\Gamma_1, \ldots, \Gamma_j$ are the contexts and $E_1, \ldots, E_n$ are the set of the type equations generated for the arguments $p_1, \ldots, p_j$ of the constructor $C_i$respectively. The arguments of the constructors are patterns themselves.

- The output type T of a constructor pattern is the data type of which it is a constructor. The type equation generated is:

$$T = D_1(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables.

- The type T_k of the argument p_k is equated with the k^{th} element of the input type of the constructor function of C_i obtained from the symbol table. The type equation generated is:

$$T_k = T_{ik} \left[D_1(A_1, \ldots, A_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\right] \quad \{1 \leq k \leq j\}$$

  where $T_{ik} \left[D_1(A_1, \ldots, A_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\right]$ represents the α-renamed k^{th} element of the input type of the constructor function of C_i.

5.4.4 Record Patterns

One can pattern match on the records of a codata type in MPL. Such patterns are called record patterns. The first rule in the Table 5.13 shows the type equations for the record patterns. The codata type, the record of which is used in the type equations, is $C_1(A_1, \ldots, A_k)$, which was defined in the Section 5.2.1 of the chapter. A description of the type equations for the record patterns is:

$$\frac{\begin{array}{c} \models\ p_1 : T_1 \quad \Big\langle \Gamma_1, E_1 \Big\rangle \\ \vdots \\ \models\ p_m : T_m \quad \Big\langle \Gamma_m, E_m \Big\rangle \end{array}}{\models \quad \begin{array}{|lcl} D_1 & : & p_1 \\ \vdots & & \vdots \\ D_m & : & p_m \end{array}^{\mathsf{RPatt}} : T \quad \Big\langle \Gamma_1 \cup \ldots \cup \Gamma_m, E_{rec} \Big\rangle} \text{ record}$$

where:

$$E_{rec} = \left\langle \exists \begin{array}{l} A_1, \ldots, A_k, \\ T_1, \ldots, T_m. \end{array} \begin{array}{l} T = C_1(A'_1, \ldots, A'_k), \\ T_1 = (T_{11}, \ldots, T_{1a}, Z_1 \to P_1)\Big[C_1(A_1, \ldots, A_k)/Z_1, A'_1/A_1, \ldots, A'_1/A_k\Big], \\ \qquad\qquad \vdots \\ T_m = (T_{m1}, \ldots, T_{mn}, Z_1 \to P_m)\Big[C_1(A_1, \ldots, A_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big], \\ E_1, \ldots, E_m \end{array} \right\rangle$$

$$\frac{\begin{array}{c} \models\ p_1 : T_1 \quad \Big\langle \Gamma_1, E_1 \Big\rangle \\ \vdots \\ \models\ p_j : T_j \quad \Big\langle \Gamma_j, E_j \Big\rangle \end{array}}{\models\ \mathsf{PPatt}\ \ (\mathsf{p_1}, \ldots, \mathsf{p_j}) : T} \text{ product}$$

$$\left\langle\ \Gamma_1 \cup \ldots \cup \Gamma_m, \left\{ \exists \begin{array}{lll} A_1, \ldots, A_k, & & T = (T_1, \ldots, T_j), \\ T_1, \ldots, T_j & . & E_1, \ldots, E_j \end{array} \right\}\ \right\rangle$$

Table 5.13: Type Equations for Record and Product Patterns

- The output type T of a record pattern is the codata type of the which the given pattern is a record. The type equation generated is:

$$T = C_1(A'_1, \ldots, A'_k)$$

where $A'_1, \ldots, A'_k$ are fresh variables.

- $\Gamma_1, \ldots, \Gamma_m$ are the contexts, and $E_1, \ldots, E_m$ are the type equations generated for the patterns $p_1, \ldots, p_m$ respectively. The patterns $p_1, \ldots, p_m$ are the patterns associated with the destructors $D_1, \ldots, D_m$ in the body of the record pattern.

- Type T_i of the pattern p_i corresponding to the destructor D_i in the record is given by the equation:

$$T_i = (T_{i1}, \ldots, T_{ij}, Z_1 \to P_i)\Big[C_1(A_1, \ldots, A_k)/Z_1, A'_1/A_1, \ldots, A'_k/A_k\Big]$$

where $T_{i1}, \ldots, T_{ij}, Z_1 \to P_i$ represents the α-renamed type of the destructor function of D_i.

5.4.5 Product Patterns

A product pattern consists of a tuple of patterns. A description of the type equations for product patterns, which is the second rule in the Table 5.13, is:

- $\Gamma_1, \ldots, \Gamma_j$ are the contexts and the $E_1, \ldots, E_j$ are the set of the type equations generated for the patterns $p_1, \ldots, p_j$ respectively.

- The output type T of a product pattern is the product of the types of the individual elements of the product pattern. This relationship is represented by the equation:

$$T = (T_1, \ldots, T_n)$$

where ( ) represents the product type which is a built in type in MPL and $T_1, \ldots, T_n$ are the types of $p_1, \ldots, p_j$ respectively.

5.5 Generating Type Equations for Function Definitions

Function definitions consist of a list of pattern phrases. Type equations for a pattern phrase are described in the Section 5.3. In Section 5.5.1, the type equations for the function defintions, which consist of a list of pattern phrases, is discussed. MPL allows the programmers to annotate a function with its expected type. Section 5.5.2 deals with the type checking of functions with an annotated type. Section 5.5.3 describes the type equations for mutually recursive function definitions.

5.5.1 Function Definitions without an Annotated Type

The first rule in the Table 5.14 provides the type equations for functions which don't have an annotated type. A description of the type equations for such function definitons is:

- The types of the different pattern phrases in the function definition should be the same. This type is also the type of the function f. The type equation generated is:

$$T_1 = T, \ldots, T_m = T$$

  where $T_1, \ldots, T_m$ are the types of the m pattern phrases and T is the type of the function f.

- $E_1, \ldots, E_m$ represent the set of type equations generated for the m pattern phrases respectively.

$$\frac{\begin{array}{l}\models\ (p_{1,1},\ldots,p_{1,n}\ \rightarrow\ t_1\ ) : T_1 \quad \left\langle E_1 \right\rangle \\ \qquad\qquad\qquad\qquad\ \vdots \\ \models\ (p_{m,1},\ldots,p_{m,n}\ \rightarrow\ t_m\ ) : T_m \quad \left\langle E_m \right\rangle\end{array}}{\begin{array}{l}\models\ \mathsf{fun\ f\ =} \\ \qquad (p_{1,1},\ldots,p_{1,n}\ \rightarrow\ t_1\ ) \\ \qquad\qquad\ \vdots \qquad\qquad\ : T \quad \left\langle \exists\, T_1,\ldots,T_m.\ \begin{array}{l} T_1 = T,\ldots,T_m = T, \\ E_1,\ldots,E_m \end{array} \right\rangle \\ \qquad (p_{m,1},\ldots,p_{m,n}\ \rightarrow\ t_m\ )\end{array}}\ \text{fun}$$

$$\frac{\begin{array}{l}\models\ \mathsf{fun\ f\ =} \\ \qquad (p_{1,1},\ldots,p_{1,n}\ \rightarrow\ t_1\ ) \\ \qquad\qquad\ \vdots \qquad\qquad\ : T \quad \left\langle E_{fun} \right\rangle \\ \qquad (p_{m,1},\ldots,p_{m,n}\ \rightarrow\ t_m\ )\end{array}}{\begin{array}{l}\models\ \mathsf{fun\ f}\ :\ \forall\ A_1,\ldots,A_k\ .\ F\ = \\ \qquad (p_{1,1},\ldots,p_{1,n}\ \rightarrow\ t_1\ ) \\ \qquad\qquad\ \vdots \qquad\qquad\ : T \\ \qquad (p_{m,1},\ldots,p_{m,n}\ \rightarrow\ t_m\ )\end{array}}\ \text{annot. fun}$$

$$\left\langle \begin{array}{lll} \forall\ A'_1,\ldots,A'_k, & T\ =\ F\ [A'_1/A_1,\ldots,A'_k/A_k], \\ \exists\ T & .\ \ E_{fun} \end{array} \right\rangle$$

Table 5.14: Type Equations for Function Definitions

Once the most general type of a function definition is inferred, the symbol table is updated with the function name and its most general type.

5.5.2 Function Definitions with an Annotated Type

The first rule of the Table 5.14 shows the type equations for a function definition f annotated with a type. The strategy used to type check the function f is: Infer the most general type of the function f without its annotated type and try to unify the inferred type with the annotated type. If the unification is successful then the annotated type of the function is right otherwise there is a discrepancy between the annotated type and the inferred type, which should result in a type error. Note that type inference yields the most general type of a function whereas the annotated type may be a specific instance of the most general type of the function. A description of the type equations for the type annotated function definitions is:

- The annotated type of the function f is:

$$\forall\ A_1, \ldots, A_k\ .\ F$$

  where F is the type of the function f and $\forall\ A_1, \ldots, A_k$ represents the type variables in the body of F.

- The type T of the function f is equated with the type F. the annotated type of the function. The type equation generated is:

$$T = F[A'_1/A_1, \ldots, A'_k/A_k]$$

  where A'_1, A'_k are fresh variables.

- E_{fun} is the set of functions generated for the function f without its annotated type.

if the annotated function type checks then the symbol table is updated with the function name and its annotated type.

5.5.3 Mutually Recursive Function Definitions

Table 5.15 provides the type equations for mutually recursive functions. The basic idea behind the rule is that the set of the mutually recursive functions are considered one function definition for the type inference process. The type equations for mutually recursive functions are generated together, which are then solved. A description of the type equations for a set of mutually recursive functions $f_1, \ldots, f_k$, shown in the Table 5.15, is:

- The functions $f_1, \ldots, f_k$ are assigned dummy types and the symbol table is expanded with the function names and the dummy types assigned to them. In this way the problem of calling a function before it is defined, as is the case in mutually recursive functions, is addressed.

- The equations for individual functions are generated and combined to get the final equation for the set of mutually recursive functions. For the function f_1 of type T having m pattern phrases of types $T_1, \ldots, T_m$ respectively, the type equations generated are:

$$T_1 = T, \ldots, T_m = T$$

  meaning that the types of all the pattern phrases should be the same which is also the type of the function. $F_1, \ldots, F_m$ are the set of type equations generated for the m pattern phrases of f_1. Similarly to f_1, the type equations are generated for all the functions in the set of mutually recursive functions.

Mutually recursive functions

$$\frac{\begin{array}{lll} \models (p_{1,1},\ldots,p_{1,n} \rightarrow t_1) : T_1 \quad \left\langle F_1 \right\rangle & & \models (q_{1,1},\ldots,q_{1,y} \rightarrow t_1) : S_1 \quad \left\langle G_1 \right\rangle \\ \vdots & & \vdots \\ \models (p_{m,1},\ldots,p_{m,n} \rightarrow t_m) : T_m \quad \left\langle F_m \right\rangle & \ldots & \models (q_{x,1},\ldots,q_{x,y} \rightarrow t_x) : S_x \quad \left\langle G_x \right\rangle \end{array}}{\begin{array}{l} \models \mathsf{defn} \\ \quad \mathsf{fun}\ \mathsf{f_1} = \\ \quad\quad (p_{1,1},\ldots,p_{1,n} \rightarrow t_1) \\ \quad\quad\quad \vdots \quad\quad\quad : T \\ \quad\quad (p_{m,1},\ldots,p_{m,n} \rightarrow t_m) \\ \\ \quad\quad \vdots \\ \\ \quad \mathsf{fun}\ \mathsf{f_k} = \\ \\ \quad\quad (q_{1,1},\ldots,q_{1,y} \rightarrow t_1) \\ \quad\quad\quad \vdots \quad\quad\quad : S \quad \left\langle \exists \begin{array}{l} T_1,\ldots,T_m \\ \quad\vdots \\ S_1,\ldots,S_x \end{array} . \begin{array}{l} T_1 = T,\ldots,T_m = T, \\ \quad\vdots \\ S_1 = S,\ldots,S_m = S, \\ F_1,\ldots,F_m, \\ \quad\vdots \\ G_1,\ldots,G_x \end{array} \right\rangle \\ \quad\quad (q_{x,1},\ldots,p_{x,y} \rightarrow t_x) \end{array}}$$

Table 5.15: Type Equations for Mutually Recursive Function Definitions

Chapter 6

Type Equations for Concurrent MPL

This chapter deals with the generation of the type equations for Concurrent MPL. Recall that the types in the concurrent world are known as channel types. The channel types are formed inductively using the built-in channel types which are listed below:

- Get
- Put
- Tensor
- Par
- Neg
- TopBot

A detailed description of the built-in channel types has been provided in the Section 3.3.2 of the third chapter. In the following sections, we will see the generation of type equations for various Concurrent MPL Constructs, process phrases, processes with and without annotated types, and mutually recursive processes. The type equation generation rules for the get and the put constructs, the split and the fork constructs, and the close and the halt constructs are motivated from [1].

$$\frac{s :: x : X, \Phi \mid \Gamma, \alpha : S \Vdash \Delta \quad \langle E \rangle}{\mathsf{get}\ \mathsf{x}\ \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists\, X, S.\, T = \mathrm{Put}(X, S), E \right\rangle}\ \text{get}$$

$$\frac{s :: x : X, \Phi \mid \Gamma \Vdash \alpha : S, \Delta \quad \langle E \rangle}{\mathsf{get}\ \mathsf{x}\ \alpha\ .s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists\, X, S.\, T = \mathrm{Get}(X, S), E \right\rangle}\ \text{get}$$

$$\frac{\Phi \vdash x : X \quad \langle E_1 \rangle \quad s :: \Phi \mid \Gamma, \alpha : S \Vdash \Delta \quad \langle E_2 \rangle}{\mathsf{put}\ \mathsf{x}\ \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists\, X, S.\, T = \mathrm{Get}(X, S), E_1, E_2 \right\rangle}\ \text{put}$$

$$\frac{\Phi \vdash x : X \quad \langle E_1 \rangle \quad s :: \Phi \mid \Gamma \Vdash \alpha : S, \Delta \quad \langle E_2 \rangle}{\mathsf{put}\ \mathsf{x}\ \alpha\ .s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists\, X, S.\, T = \mathrm{Put}(X, S), E_1, E_2 \right\rangle}\ \text{put}$$

Table 6.1: Type Equations for the get and the put Constructs

Type equations for the various concurrent MPL constructs are generated in the process context. A process context consists of a sequential context Φ, a input channel context Γ and an output channel context Δ and is represented as:

$$\Phi \mid \Gamma \Vdash \Delta$$

The sequential context contains the list of pairs of variables and their associated types. The channel contexts are a list of pair of channels with their associated channel types.

6.1 The get and the put Constructs

The get construct gets a value from a channel and the put construct puts a value on a channel. The first two rules in Table 6.1 show the type equation rule for the get construct. A description of the equations is:

- The channel type T of the get construct in get x α is the channel type of α, which can

be looked up from the channel contexts. The polarity of the channel α depends on the channel context, input or output, in which the channel name is present.

- The channel type T of `get x` α for an input polarity channel α is given by the equation:

$$T \;=\; \mathrm{Put}(X, S)$$

  where X is the type of the variable x, and S is the channel type corresponding to those concurrent MPL constructs in s which operate on the channel α and are below the current `get` construct in the process block.

- The channel type T of `get x` α for an output polarity channel α is given by the equation:

$$T \;=\; \mathrm{Get}(X, S)$$

- The set of type equations for the concurrent MPL commands which are below the current `get` construct in the process body are represent by E. These equations are generated in a modified process context: the sequential context is extended by adding the variable x with its type X and the input or the output channel context is modified by associating the channel α with the type S instead of the type T.

A description of the type equations for the `put` construct, shown by the last two rules in the Table 6.1, is:

- The channel type T of the `put` construct in `put x` α is the channel type of α which can be looked up from the channel contexts.

- The channel type T of `put x` α for an input polarity channel α is given by the equation:

$$T \;=\; \mathrm{Get}(X, S)$$

where X is the type of the term x, and S is the channel type corresponding to those concurrent MPL constructs in s which operate on the channel α and are below the current put construct in the process block.

- The channel type T of put x α for an output polarity channel α is given by the equation:

$$T \;=\; Put(X, S)$$

- The set of type equations for the concurrent MPL commands below the current put construct in the process block are represent by E. These type equations are generated in a modified process context where one of the channel contexts, input or output, has α associated with S instead of T.

- E_1 is the set of type equations generated for the sequential term t in the sequential context Φ.

6.2 The split and the fork Constructs

The split and the fork constructs come in a pair. The split construct splits a channel into two channels and the fork construct creates two processes by forking the current process on the channel which is split.

A description of the type equation for the split construct, shown in the first two rules in the Table 6.2, is:

- If α is an input polarity channel of channel type T which splits to produce two input polarity channels α_1, and α_2 of channel types T_1, and T_2 respectively, then the channel type of α is the *tensor* of the channel types of α_1 and α_2 represented by the equation:

$$T \;=\; T_1 \;(*)\; T_2$$

$$\frac{s :: \Phi \mid \Gamma, \alpha_1 : T_1, \alpha_2 : T_2 \Vdash \Delta \quad \langle E \rangle}{\mathsf{split}\ \ \alpha\ \ (\alpha_1, \alpha_2).s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists\, T_1, T_2.\, T = T_1\ (*)\ T2, E \right\rangle}\ \text{split}$$

$$\frac{s :: \Phi \mid \Gamma \Vdash \alpha_1 : T_1, \alpha_2 : T_2, \Delta \quad \langle E \rangle}{\mathsf{split}\ \ \alpha\ \ (\alpha_1, \alpha_2).s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists\, T_1, T_2.\, T = T_1\ (+)\ T_2, E \right\rangle}\ \text{split}$$

$$\frac{s_1 :: \Phi \mid \Gamma_1, \alpha_1 : T_1 \Vdash \Delta_1 \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \alpha_2 : T_2, \Gamma_2 \Vdash \Delta_2 \quad \langle E_2 \rangle}{\begin{array}{l}\mathsf{fork}\ \ \alpha \\ \mathsf{as} \left| \begin{array}{lcll} \alpha_1 & \rightarrow & s_1 & :: \Phi \mid \Gamma_1, \alpha : T, \Gamma_2 \Vdash \Delta_1, \Delta_2 \quad \left\langle \exists\, T_1, T_2.\, T = T_1\ (+)\ T_2, E_1, E_2 \right\rangle \\ \alpha_2 & \rightarrow & s_2 & \end{array} \right.\end{array}}\ \text{fork}$$

$$\frac{s_1 :: \Phi \mid \Gamma_1 \Vdash \Delta_1, \alpha_1 : T_1 \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \Gamma_2 \Vdash \alpha_2 : T_2, \Delta_2 \quad \langle E_2 \rangle}{\begin{array}{l}\mathsf{fork}\ \ \alpha \\ \mathsf{as} \left| \begin{array}{lcll} \alpha_1 & \rightarrow & s_1 & :: \Phi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \alpha : T, \Delta_2 \quad \left\langle \exists\, T_1, T_2.\, T = T_1\ (*)\ T_2, E_1, E_2 \right\rangle \\ \alpha_2 & \rightarrow & s_2 & \end{array} \right.\end{array}}\ \text{fork}$$

Table 6.2: Type Equations for the split and the fork Constructs

- Similarly, splitting an output polarity channel results in the *par* channel type. The type equation generated is:

$$T \ = \ T_1\ (+)\ T_2$$

- The channel context is extended with the newly created channel names α_1 and α_2 along with their respective channel types. If the original channel α was an input polarity channel then the input channel context is enhanced else the output channel context is extended. E represents the set of type equations generated for the process commands s in the extended channel context. s represents the process commands which are working on the channel α, and are below the split construct in the process block.

A description of the type equations for the fork construct, shown in the last two rules of Table 6.2, is:

- Forking on an input polarity channel α of channel type T, which expects the channel α to split and produce two new input polarity channels α_1 and α_2 of channel types T_1 and T_2 repsectively, results in the built-in channel type *par*. This generates the type equation:

$$T \;=\; T_1 \;(+)\; T_2$$

- Similarly, forking on an output polarity channel results in the channel type *par*. The type equation generated is:

$$T \;=\; T_1 \;(*)\; T_2$$

- The channel names α_1 and α_2 along with their respective channel types separately extend the output or the input channel context generating two new contexts depending on the polarity of α. The channel context extended with α_1 is used to generate the type equations for the process commands s_1, the process corresponding to the α_1 branch. E_1 represents the set of type equations generated for s_1 in the channel context extended by α_1 and its channel type T_1. Similarly, E_2 represents the set of type equations generated for s_2 in the channel context extended by α_2 and its channel type T_2.

6.3 The hput and the hcase Constructs

The hput construct puts a handle/cohandle H on a channel α. The (co)protocol of the (co)handle H becomes the channel type of α. The hcase construct branches on the (co)handles of a protocol. One hputs a handle on an output polarity channel and hcases on the handles of a protocol on an input polarity channel. Alternatively, one hputs a cohandle on an input polarity channel and hcases on cohandles on an output polarity channel.

$$\frac{s :: \Phi \mid \Gamma \Vdash \alpha : S, \Delta \;\; \langle E \rangle}{\mathsf{hput}\ H_i\ \mathsf{on}\ \alpha\ .s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta}\ \text{hput}$$

$$\left\langle \exists\, S, A'_1, \ldots, A'_k\ .\ \begin{array}{l} T = P(A'_1, \ldots, A'_k), \\ S = T_i\big[P(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\big], \\ E \end{array} \right\rangle$$

$$\frac{s :: \Phi \mid \Gamma, \alpha : S \Vdash \Delta \;\; \langle E \rangle}{\mathsf{hput}\ H_i\ \mathsf{on}\ \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta}\ \text{hput}$$

$$\left\langle \exists\, S, A'_1, \ldots, A'_k\ .\ \begin{array}{l} T = P'(A'_1, \ldots, A'_k), \\ S = T_i\big[P(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\big], \\ E \end{array} \right\rangle$$

$$\frac{c_1 :: \Phi \mid \Gamma, \alpha : S_1 \Vdash \Delta \;\; \langle E_1 \rangle, \ldots, c_n :: \Phi \mid \Gamma, \alpha : S_n \Vdash \Delta \;\; \langle E_n \rangle}{\begin{array}{l} \mathsf{hcase}\ \alpha \\ \quad \mathsf{of} \left| \begin{array}{lcl} H_1 & \to & c_1 \\ \vdots & \vdots & \vdots \\ H_n & \to & c_n \end{array} \right. :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \end{array}}\ \text{hcase}$$

$$\left\langle \exists\ \begin{array}{l} A'_1, \ldots, A'_k, \\ S_1, \ldots, S_n \end{array}\ .\ \begin{array}{l} T = P\ \big(A'_1, \ldots, A'_k\big), \\ S_1 = T_1\big[P(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\big], \\ \qquad\qquad \vdots \\ S_n = T_n\big[P(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\big], \\ E_1, \ldots, E_n \end{array} \right\rangle$$

$$\frac{c_1 :: \Phi \mid \Gamma \Vdash \alpha : S_1, \Delta \;\; \langle E_1 \rangle, \ldots, c_n :: \Phi \mid \Gamma, \Vdash \alpha : S_n, \Delta \;\; \langle E_n \rangle}{\begin{array}{l} \mathsf{hcase}\ \alpha \\ \quad \mathsf{of} \left| \begin{array}{lcl} H_1 & \to & c_1 \\ \vdots & \vdots & \vdots \\ H_n & \to & c_n \end{array} \right. :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \end{array}}\ \text{hcase}$$

$$\left\langle \exists\ \begin{array}{l} A'_1, \ldots, A'_k, \\ S_1, \ldots, S_n \end{array}\ .\ \begin{array}{l} T = P'\ \big(A'_1/A^1_1, \ldots, A'_k/A^1_k\big), \\ S_1 = T_1\big[P(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\big], \\ \qquad\qquad \vdots \\ S_n = T_n\big[P(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\big], \\ E_1, \ldots, E_n \end{array} \right\rangle$$

Table 6.3: Type Equations for the hput and the hcase Constructs

6.3.1 Populating the symbol table for (Co)Protocol Declarations

Consider a protocol P with handles $H_1, \ldots, H_n$ defined below:

$$\begin{aligned} &\mathsf{protocol}\ P(A_1, \ldots, A_k)\ \Rightarrow\ C = \\ &\quad H_1 : T_1 \Rightarrow C \\ &\quad \vdots \quad \vdots \quad \vdots \\ &\quad H_n : T_n \Rightarrow C \end{aligned}$$

where the protocol P is polymorphic in type variables $A_1, \ldots, A_k$. The channel type of a handle is obtained by replacing the state variable C with the protocol name $P(A'_1, \ldots, A'_k)$, where $A'_1, \ldots, A'_k$ are fresh variables used for $A_1, \ldots, A_k$ respectively. The channel type of the handle H_i, held in symbol table, is:

$$(\forall\ A'_1, \ldots, A'_k\ (\forall\ C, A_1, \ldots, A_k\ .\ T_i \Rightarrow C)\ (P(A'_1, \ldots, A'_k))\ A'_1, \ldots, A'_k$$

which can be β-reduced to

$$(T_i \Rightarrow C)\ \left[P(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\right]$$

Consider a coprotocol P' with cohandles $H_1\ \ldots\ H_n$ defined as:

$$\begin{aligned} &\mathsf{coprotocol}\ P'(A_1, \ldots, A_k)\ \Rightarrow\ C = \\ &\quad H_1 : C \Rightarrow T_1 \\ &\quad \vdots \quad \vdots \quad \vdots \\ &\quad H_n : C \Rightarrow T_n \end{aligned}$$

Like with handles, the types of the cohandles are held in the symbol table. The β-reduced

type of a cohandle H_i is given by:

$$(T_i \Rightarrow C)\ \left[P'(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\right]$$

6.3.2 Description of the Type Equations

A description of the type equations for the hput construct in the case where a handle is hput on an output channel, which is the first rule in the Table 6.3, is:

- The type T of the channel α is the protocol of which H_i is a handle of. The type equation generated is:

  $$T \ = \ P(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables.

- The channel type S of the constructs in s which work on α, is given by the type equation:

  $$S \ = \ T_i\left[P'(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k\right]$$

  where T_i is the input type of the handle H_i.

- E is the set of type equations generated in a process context where the output channel context is modified to have S instead of T as the channel type associated with the channel α.

The type equation for the case when a cohandle is hput on an input polarity channel is symmetric to the case when a handle is hput on an output polarity channel.

A description of the type equations for the case when the handles of a protocol are cased on an input polarity channel, which is the third rule in the Table 6.3, is:

- The type T of the channel α is the protocol of the handles which are hcased on. The type equation generated is:

$$T \;=\; P(A'_1, \ldots, A'_k)$$

- The channel type S_i for α corresponding to the concurrent MPL constructs in C_i is given by:

$$S_i \;=\; T_i[P'(A'_1, \ldots, A'_k)/C, A'_1/A_1, \ldots, A'_k/A_k]$$

- $E_1, \ldots, E_n$ represent the set of type equations for the concurrent MPL commands $c_1, \ldots, c_n$, which are the set of commands corresponding to the handles $H_1, \ldots, H_n$ respectively. E_i, for $1 \leq i \leq n$, is generated in the process context where the output channel context is modified such that the type S_i is associated with the channel α instead of the type T.

Handling the cohandles of a coprotocol on an output polarity channel is symmetric to the above case.

6.4 The close and the halt Constructs

The close and the halt constructs are used to close the channels used in an MPL process. All the channels must be closed before halting an MPL process. The halt construct occurs as the last command, however the close construct has other concurrent commands below it in the process block. A description of the type equations for the close constructs, shown in the first two rows of the Table 6.4 , is:

- Closing an input or an output channel α of type T results in the built-in channel type

$$\frac{s :: \Phi \mid \Gamma \Vdash \Delta \quad \langle E \rangle}{\mathsf{close}\ \ \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle T = \text{TopBot}, E \right\rangle}\ \text{close}$$

$$\frac{s :: \Phi \mid \Gamma \Vdash \Delta \quad \langle E \rangle}{\mathsf{close}\ \ \alpha\ .s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle T = \text{TopBot}, E \right\rangle}\ \text{close}$$

$$\frac{\phi :: \Phi \mid \phi \Vdash \phi}{\mathsf{halt}\ \ \alpha\ :: \Phi \mid \alpha : T \Vdash \phi \quad \left\langle T = \text{TopBot} \right\rangle}\ \text{halt}$$

$$\frac{\phi :: \Phi \mid \phi \Vdash \phi}{\mathsf{halt}\ \ \alpha\ :: \Phi \mid \phi \Vdash \alpha : T \quad \left\langle T = \text{TopBot} \right\rangle}\ \text{halt}$$

Table 6.4: Type Equations for the close and the halt Constructs

TopBot. The type equation generated is:

$$T \ = \ \text{TopBot}$$

- E is the set of type equations generated for the subsequent process commands s in the process block.

The type equations for the halt construct are shown in the last two rows of the Table 6.4. Halting an input or an output channel α of type T results in the built-in channel type TopBot. The type equation generated is:

$$T \ = \ \text{TopBot}$$

$$\frac{}{\alpha \mid = \mid \beta :: \Phi \mid \alpha : T_1 \Vdash \beta : T_2 \quad \langle T_1 = T_2 \rangle}\ \text{id}$$

$$\frac{}{\alpha \mid = \mid \beta :: \Phi \mid \beta : T_2 \Vdash \alpha : T_1 \quad \langle T_1 = T_2 \rangle}\ \text{id}$$

$$\frac{}{\alpha \mid = \mid \mathtt{neg}(\beta) :: \Phi \mid \alpha : T_1, \beta : T_2 \Vdash \phi \quad \langle T_1 = \mathrm{Neg}(T_2) \rangle}\ \text{neg}$$

$$\frac{}{\alpha \mid = \mid \mathtt{neg}(\beta) :: \Phi \mid \phi \Vdash \alpha : T_1, \beta : T_2 \quad \langle T_1 = \mathrm{Neg}(T_2) \rangle}\ \text{neg}$$

Table 6.5: Type Equation for the id and the neg Constructs

6.5 The id and the neg Constructs

The type equations generated for the id and the neg constructs are shown in Table 6.5. The id command is used to equate two channels of opposite polarities. The type equation is:

$$T_1 \ = \ T_2$$

The neg construct equates two channels having the same polarity. The type equation is:

$$T_1 \ = \ \mathrm{Neg}(T_2)$$

6.6 The plug Construct

The plug construct connects two processes along a channel. A description of the type equations for the plug construct, shown in the first row of the Table 6.6 is:

- Channel α is plugged between two processes s_1 and s_2. α works as an output channel for s_1 and an input channel for s_2.
- E_1 is the set of type equations generated for s_1. The output channel context in which

$$\frac{s_1 :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta_1 \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \Gamma, \alpha : S \Vdash \Delta_2 \quad \langle E_2 \rangle}{\mathsf{plug}\ \ \alpha\ (\mathsf{s_1}, \mathsf{s_2}) :: \Phi \mid \Gamma \Vdash \Delta \quad \left\langle \exists T, S\ .\ T = S, E_1, E_2 \right\rangle}\ \text{plug}$$

$$\frac{\begin{array}{c}\Phi \vdash x_1 : T_1 \quad \langle E_1 \rangle, \\ \vdots \\ \Phi \vdash x_n : T_n \quad \langle E_n \rangle\end{array}}{\mathsf{p}\ (\mathsf{x_1}, \ldots \mathsf{x_n} | \alpha_1, \ldots, \alpha_\mathsf{p} \rightarrow \beta_1 \ldots \beta_\mathsf{q}) :: \Phi \mid \Gamma, \alpha_1 : A_1 \ldots \alpha_p : A_p \Vdash \beta_1 : B_1, \ldots, \beta_q : B_q, \Delta}\ \text{pcall}$$

$$\left\langle \exists \begin{array}{ll} X_1, \ldots, X_n, & X_1 = T_1\left[V_1'/V_1, \ldots, V_k'/V_k\right], \ldots, X_n = T_n\left[V_1'/V_1, \ldots, V_k'/V_k\right], \\ Y_1, \ldots, Y_p\ . & Y_1 = A_1\left[V_1'/V_1, \ldots, V_k'/V_k\right], \ldots, Y_p = A_p\left[V_1'/V_1, \ldots, V_k'/V_k\right], \\ Z_1, \ldots, Z_q, & Z_1 = B_1\left[V_1'/V_1, \ldots, V_k'/V_k\right], \ldots, Z_q = B_q\left[V_1'/V_1, \ldots, V_k'/V_k\right], \\ V_1', \ldots, V_k' & E_1, \ldots, E_n \end{array} \right\rangle$$

Table 6.6: Type Equations for the process call, and the plug

E_1 is generated is extended with the channel α with its type S. The other two contexts, namely the sequential context and the input channel contexts remain unchanged.

- E_2 is the set of type equations generated for s_2. The output channel context in which E_2 is generated is extended with the channel α with its type T. The other two contexts, namely the sequential context and the input channel contexts remain unchanged.

- The type of the channel α used in s_1 and s_2 should unify. The type equation generated is:

$$T = S$$

6.7 Process Call

The process call construct is used to call an already defined process with arguments. When a process call is made, the type inference algorithm uses the symbol table for the following:

- Lookup the process name in the symbol table to verify if a valid process is being called. If the process name is not present in the symbol table, then an error message is reported.
- If the process name is found in the symbol table, then its corresponding type is found and used in the generation of type equations for the process call.

Consider a process defintion p which has n sequential paramaters of types $T_1, \ldots, T_n$, p input polarity channels of channel types $A_1, \ldots, A_p$ and q output polarity channels of channel types $B_1, \ldots, B_q$. The type of the process is represented as:

$$p: \ \forall V_1, \ldots, V_k.\ T_1, \ldots, T_n \mid A_1, \ldots, A_p \ \Rightarrow \ B_1, \ldots, B_q$$

where $V_1, \ldots, V_k$ represent the union of all the type variables present in the process type.

The sequential types and the channel types are α-renamed before using them in a type equation in order to avoid name clash between the variables used in the type expressions and the variables already used in the type equation. The α-renamed sequential types, input channel types, and output channel types are:

$$\begin{aligned}
&T_i \left[V_1'/V_1, \ldots, V_k'/V_k\right] \ \ \{1 \leq i \leq n\}(\textsf{sequential type}) \\
&A_j \left[V_1'/V_1, \ldots, V_k'/V_k\right] \ \ \{1 \leq j \leq p\}(\textsf{input channel type}) \\
&B_l \left[V_1'/V_1, \ldots, V_k'/V_k\right] \ \ \{1 \leq l \leq q\}(\textsf{output channel type})
\end{aligned}$$

where $\left[V_1'/V_1, \ldots, V_k'/V_k\right]$ represents a list of substitutions to be made inside the type expressions T_i, A_j, and B_l. A description of the type equations for the process call construct,

shown in the last rule in the Table 6.6, is:

- A process call is the last command in a process block.
- The type X_i of x_i, which is i^{th} sequential argument, is equated with the corresponding element of the process type of p obtained from the symbol table. The type equation generated is:

$$X_i = T_i \left[V_1'/V_1, \ldots, V_k'/V_k\right] \quad \{1 \leq i \leq n\}$$

  where $T_i \left[V_1'/V_1, \ldots, V_k'/V_k\right]$ represents the α-renamed i^{th} sequential type of the process type of p.

- The type Y_j of α_j, which is the name of the j^{th} input polarity channel, is given by the equation:

$$Y_j = A_j \left[V_1'/V_1, \ldots, V_k'/V_k\right] \quad \{1 \leq j \leq p\}$$

- The type Z_l of β_l, which is the name of the l^{th} output polarity channel, is given by the equation:

$$Z_l = B_l \left[V_1'/V_1, \ldots, V_k'/V_k\right] \quad \{1 \leq j \leq p\}$$

6.8 Generating Type Equations for a Process Phrase

A process body consists of a sequence of proces phrases. Recall that a process phrase consists of sequential patterns, input channel names and output channel names. A description of the type equations for the process phrase, shown in the Table 6.7, is:

- $(x_1, \ldots, x_n \mid i_1, \ldots, i_p \;\Rightarrow\; o_1, \ldots, o_q \rightarrow\; c\;)$ represents a process phrase of type T. $x_1, \ldots, x_n$ represent the n patterns of the process phrase of types $T_1, \ldots, T_n$ respectively,

$i_1, \ldots, i_p$ represent the p input channels of channel types $A_1, \ldots, A_p$ respectively, and $o_1, \ldots, o_q$ represent the q output channels of channel types $B_1, \ldots, B_q$ respectively. The type T of the process phrase is:

$$T \;=\; T_1, \ldots, T_n \;\mid\; A_1, \ldots, A_p \;\Rightarrow\; B_1, \ldots, B_q$$

where $\mid$ separates the sequential types from the input channel types and $\Rightarrow$ separates the input channel types from the output channel types.

- The type equation generation process for the process phrase starts with with a sequential context Φ and empty input and output channel contexts.

- Every sequential pattern x_i, for $1 \leq i \leq n$, generates a sequential context Φ_i and a set of type equations E_i in an empty sequential context represented by the symbol $\models$.

- The set of type equations E are generated for the process commands c, in a process context generated as follows:

  - The sequential context is obtained by the union of the original sequential context Φ with the n sequential contexts, $\Phi_1, \ldots, \Phi_n$, generated for the n patterns in the process phrase.

  - The input channel context is obtained by adding the input channels $i_1, \ldots, i_p$ and their respective types $A_1, \ldots, A_p$ to the empty input channel context.

  - The output channel context is obtained by adding the output channels $o_1, \ldots, o_q$ and their respective types $B_1, \ldots, B_q$ to the empty output channel context.

$$\frac{\begin{array}{l} \models\ x_1 : T_1 \quad \left\langle \Phi_1, E_1 \right\rangle \\ \qquad \vdots \\ \models\ x_n\ : T_n\ \left\langle \Phi_n, E_n \right\rangle \qquad c\ ::\ \Phi\ \cup\ \Phi_1 \cup, \ldots, \cup\ \Phi_n\ \mid\ \Gamma\ \Vdash\ \Delta\ \left\langle E \right\rangle \end{array}}{\Phi \mid \phi \Vdash \phi\ \vdash\ (x_1, \ldots, x_n \mid i_1, \ldots, i_p\ \Rightarrow\ o_1, \ldots, o_q \rightarrow\ c\ ) : T}\ \text{procPhr}$$

$$\left\langle \exists \begin{array}{l} T_1, \ldots, T_n \\ A_1, \ldots, A_p\ . \\ B_1, \ldots, B_q \end{array} \begin{array}{l} T = T_1, \ldots, T_n\ \mid\ A_1, \ldots, A_p\ \Rightarrow\ B_1, \ldots, B_q, \\ E, E_1, \ldots, E_n \end{array} \right\rangle$$

where Γ, and Δ used above are defined as:

$$\begin{aligned} \Gamma\ &=\ i_1 : A_1, \ldots i_p : A_p \\ \Delta\ &=\ o_1 : B_1, \ldots, o_q : B_q \end{aligned}$$

Table 6.7: Type Equations for the Process Phrase

6.9 Generating Type Equations for a Process Definition

This section discusses the type equations for process definitions with and without annotated types and mutually recursive process definitions. The generation of type equations for a process starts in an empty process context comprising of empty sequential context, empty input channel context, and empty output channel context represented as:

$$\phi\ \mid\ \phi\ \Vdash\ \phi.$$

In the generation of type equations for processes, an empty process context is represented as $\models$.

6.9.1 Process Definition Without an Annotated Type

The first row of the Table 6.8 represents the type equations generated for a process p not annotated with a process type. A description of these rules is:

- The type T of the process is the type of the individual pattern phrases. In a well typed process, the types of all the process phrases, $T_1, \ldots, T_m$, are the same. The type equations generated are: $T_1 = T, \ldots, T_m = T$.

- $E_1, \ldots, E_m$ represent the set of the type equations generated for the m process phrases.

Once the most general type of a process is inferred, the process name and its most general type is inserted into the symbol table from where it can be looked up.

6.9.2 Process Definition With an Annotated Type

The second row of the Table 6.8 represents the type equations generated for a process p with an annotated type. The strategy used to type check p is: Infer the most general type of p without its annotated type and try to unify the inferred type with the annotated type. If the unification is possible then the annotated type of the process is right otherwise there is a type error. Note that type inference yields the most general type of a process whereas the annotated type may be a specific instance of the most general type. A description of the type equations is:

- The annotated type for the process p is shown as $\forall\ A_1, \ldots, A_k\ .P$ where P is the type of the process p and $A_1, \ldots, A_k$ represents the set of type variables present in the body of P.

- The type T of the process is equated to the annotated type P. The type equation generated is:

$$T \ = \ P[A'_1/A_1, \ldots, A'_k/A_k]$$

$$\frac{\begin{array}{l} \models\ (x_{1,1},\ldots,x_{1,n}\ \mid\ i_{1,1},\ldots,i_{1,p}\ \Rightarrow\ o_{1,1},\ldots,o_{1,q}\rightarrow\ c_1\ ) : T_1 \quad \left\langle E_1 \right\rangle \\ \qquad\vdots \qquad\qquad\vdots \qquad\qquad\vdots \\ \models\ (x_{m,1},\ldots,x_{m,n}\ \mid\ i_{m,1},\ldots,i_{m,p}\ \Rightarrow\ o_{m,1},\ldots,o_{m,q}\rightarrow\ c_m\ ) : T_m \quad \left\langle E_m \right\rangle \end{array}}{\begin{array}{l} \models \left\{ \begin{array}{l} \mathsf{proc}\ \ \mathsf{p}\ = \\ \qquad (x_{1,1},\ldots,x_{1,n}\ \mid\ i_{1,1},\ldots,i_{1,p}\ \Rightarrow\ o_{1,1},\ldots,o_{1,q}\rightarrow\ c_1\ ) \\ \qquad\qquad\vdots \qquad\qquad\vdots \qquad\qquad\vdots \\ \qquad (x_{m,1},\ldots,x_{m,n}\ \mid\ i_{m,1},\ldots,i_{m,p}\ \Rightarrow\ o_{m,1},\ldots,o_{m,q}\rightarrow\ c_m\ ) \end{array} \right\} : T \\ \\ \left\langle \exists\ T_1,\ldots,T_m.\ \begin{array}{l} T_1 = T,\ldots,T_m = T, \\ E_1,\ldots,E_m \end{array} \right\rangle \end{array}}\ \text{proc}$$

$$\frac{\begin{array}{l} \models \left\{ \begin{array}{l} \mathsf{proc}\ \ \mathsf{p}\ = \\ \qquad (x_{1,1},\ldots,x_{1,n}\ \mid\ i_{1,1},\ldots,i_{1,p}\ \Rightarrow\ o_{1,1},\ldots,o_{1,q}\rightarrow\ c_1\ ) \\ \qquad\qquad\vdots \qquad\qquad\vdots \qquad\qquad\vdots \\ \qquad (x_{m,1},\ldots,x_{m,n}\ \mid\ i_{m,1},\ldots,i_{m,p}\ \Rightarrow\ o_{m,1},\ldots,o_{m,q}\rightarrow\ c_m\ ) \end{array} \right\} : T \quad \left\langle E_{proc} \right\rangle \end{array}}{\begin{array}{l} \models \left\{ \begin{array}{l} \mathsf{proc}\ \ \mathsf{p}\ :\ \forall\ A_1,\ldots,A_k\ .\ P\ = \\ \qquad (x_{1,1},\ldots,x_{1,n}\ \mid\ i_{1,1},\ldots,i_{1,p}\ \Rightarrow\ o_{1,1},\ldots,o_{1,q}\rightarrow\ c_1\ ) \\ \qquad\qquad\vdots \qquad\qquad\vdots \qquad\qquad\vdots \\ \qquad (x_{m,1},\ldots,x_{m,n}\ \mid\ i_{m,1},\ldots,i_{m,p}\ \Rightarrow\ o_{m,1},\ldots,o_{m,q}\rightarrow\ c_m\ ) \end{array} \right\} : T \\ \\ \left\langle \begin{array}{ll} \forall\ A'_1,\ldots,A'_k, & T\ =\ P\ [A'_1/A_1,\ldots,A'_k/A_k], \\ \exists\ T & .\ \ E_{proc} \end{array} \right\rangle \end{array}}\ \text{annot. proc}$$

Table 6.8: Type Equations for Process Definitions

where $A'_1, \ldots, A'_k$ are fresh variables.

- E_{proc} represents the set of type equations generated for the process p without its annotated type.

6.9.3 Mutually Recursive Process Definitions

Table 6.9 shows the type equations for mutually recursive process definitions. The idea is that the set of mutually recursive processes are considered one process for the purpose of type inference, i.e. the type equations for mutually recursive processes are generated and solved together. A description of the type equations for a set of mutually recursive processes $p_1, \ldots, p_k$ is:

- The processes $p_1, \ldots, p_k$ are assigned dummy types which are then inserted in the symbol table. In this way the problem of calling a process before it is defined, as is the case with mutually recursive processes, is addressed.

- The equations for individual processes are generated and combined to get the final equations for the set of mutually recursive processes.

$$
\begin{array}{l}
\models\ (x_{1,1,1},\ldots,x_{1,1,n}\ \mid\ i_{1,1,1},\ldots,i_{1,1,a}\ \Rightarrow\ o_{1,1,1},\ldots,o_{1,1,b}\rightarrow\ c_{1,1}\ ) : T_1 \quad \left\langle F_1 \right\rangle \\
\qquad\vdots \qquad\qquad\vdots \qquad\qquad\qquad\vdots \\
\models\ (x_{1,m,1},\ldots,x_{1,m,n}\ \mid\ i_{1,m,1},\ldots,i_{1,m,a}\ \Rightarrow\ o_{1,m,1},\ldots,o_{1,m,b}\rightarrow\ c_{1,m}\ ) : T_m \quad \left\langle F_m \right\rangle \\
\qquad\vdots \qquad\qquad\vdots \qquad\qquad\qquad\vdots \\
\models\ (x_{k,1,1},\ldots,x_{k,1,v}\ \mid\ i_{k,1,1},\ldots,i_{k,1,w}\ \Rightarrow\ o_{k,1,1},\ldots,o_{k,1,y}\rightarrow\ c_{k,1}\ ) : S_1 \quad \left\langle G_1 \right\rangle \\
\qquad\vdots \qquad\qquad\vdots \qquad\qquad\qquad\vdots \\
\models\ (x_{k,u,1},\ldots,x_{k,u,v}\ \mid\ i_{k,u,1},\ldots,i_{k,u,w}\ \Rightarrow\ o_{k,u,1},\ldots,o_{k,u,w}\rightarrow\ c_{k,u}\ ) : S_u \quad \left\langle G_u \right\rangle
\end{array}
$$

$$
\models\ \mathsf{defn}
\left\{
\begin{array}{l}
\mathsf{proc}\ \ \mathsf{p_1}\ = \\
\quad (x_{1,1,1},\ldots,x_{1,1,n}\ \mid\ i_{1,1,1},\ldots,i_{1,1,a}\ \Rightarrow\ o_{1,1,1},\ldots,o_{1,1,b}\rightarrow\ c_{1,1}\ ) \\
\qquad\vdots \qquad\qquad\vdots \qquad\qquad\qquad\vdots \\
\quad (x_{1,m,1},\ldots,x_{1,m,n}\ \mid\ i_{1,m,1},\ldots,i_{1,m,a}\ \Rightarrow\ o_{1,m,1},\ldots,o_{1,m,b}\rightarrow\ c_{1,m}\ )
\end{array}
\right\} : T
$$

$$
\vdots
$$

$$
\left\{
\begin{array}{l}
\mathsf{proc}\ \ \mathsf{p_k}\ = \\
\quad (x_{k,1,1},\ldots,x_{k,1,v}\ \mid\ i_{k,1,1},\ldots,i_{k,1,w}\ \Rightarrow\ o_{k,1,1},\ldots,o_{k,1,y}\rightarrow\ c_{k,1}\ ) \\
\qquad\vdots \qquad\qquad\vdots \qquad\qquad\qquad\vdots \\
\quad (x_{k,u,1},\ldots,x_{k,u,v}\ \mid\ i_{k,u,1},\ldots,i_{k,u,w}\ \Rightarrow\ o_{k,u,1},\ldots,o_{k,u,w}\rightarrow\ c_{k,u}\ )
\end{array}
\right\} : S
$$

$$
\left\langle \exists \begin{array}{l} T_1,\ldots,T_m \\ \quad\vdots \\ S_1,\ldots,S_u\ . \end{array} \begin{array}{l} T_1 = T,\ldots,T_m = T, \\ \qquad\vdots \\ S_1 = S,\ldots,S_m = S, \\ F_1,\ldots,F_m, \\ \quad\vdots \\ G_1,\ldots,G_u \end{array} \right\rangle
$$

Table 6.9: Type Equations for Mutually Recursive Process Definitions

Chapter 7

Compilation of Pattern-Matching

Once an MPL program type checks, the next step in the compilation of the program is the conversion of its AST to Core MPL. The compilation of the AST to the Core MPL happens in 2 steps:

- **Compilation of Pattern-Matching** - This step translates the pattern-matching syntax to MPL without patterns. This is the first step towards the conversion of the AST of MPL programs to core MPL.
- **Lambda Lifting Tranformation** - MPL allows local function definitions. However, Core MPL programs can not have local function defintions. Lambda Lifting transformation gets rid of the local function defintions in MPL programs by lifting the local function to the global scope. The lambda lifting transformation is dicussed in the next chapter.

In this chapter, the algorithm for compilation of pattern-matching is discussed. A *pattern-matching compiler algorithm* was given by Lennart Augustsson [14]. The version of the algorithm described in the thesis and used in MPL's implementation is due to Geoff Barrett and Philip Wadler [15]. However, the algorithm in its original form did not handle the record or the product patterns. The algorithm, therefore was modified to work with record and product patterns.

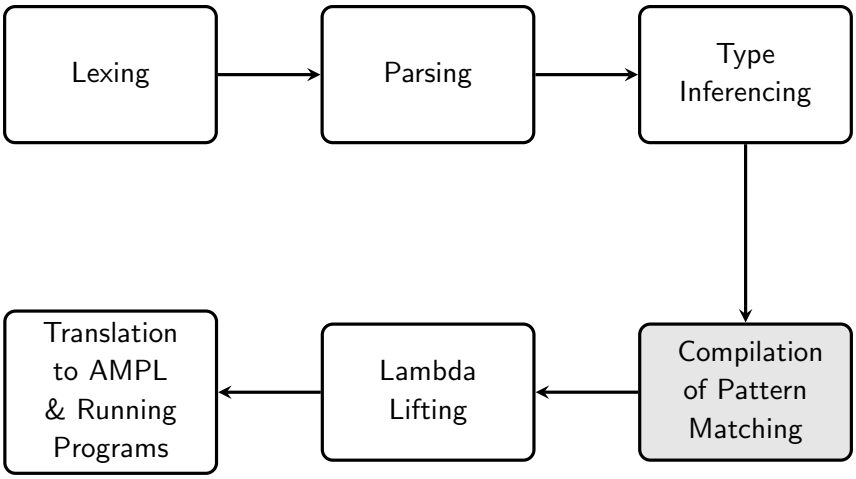

Figure 7.1: Compilation Stages : Compilation of Pattern Matching

7.1 Examples of Pattern-Matching in MPL

A function f defined with m pattern phrases can be represented as:

$$
\begin{aligned}
f = & \\
& p_{1,1}, \ldots, p_{1,n} \rightarrow t_1 \\
& \ldots \\
& p_{m,1}, \ldots, p_{m,n} \rightarrow t_m
\end{aligned}
$$

Recall that a pattern phrase consists of a list of patterns and the corresponding term. In the above function definition a pattern $p_{i,j}$ can be one of the following:

- Don't Care Pattern
- Variable Pattern
- Record Pattern
- Product Pattern

- Constructor Pattern

A function can have a mixture of these patterns in any given pattern phrase. The constructor, record and product patterns may have other patterns in their body. Don't care patterns can be thought as a special case of variable patterns in which the variables in the patterns of the pattern phrase are not used in the term of the pattern phrase. In the further discussion of the *pattern-matching compilation algorithm*, don't care patterns are not explicitly discussed as their compilation scheme is the same as that of variable patterns.

The pattern-matching compilation algorithm gets rid of the constructor, product and record patterns, including the nested patterns, in a function body. The final result of the algorithm is a function body that contains only one pattern phrase consisting of just variable patterns and the corresponding term.

Table 7.1 provides examples of functions defined with various patterns: the functions are `append`, `pairFun`, and `fstOfHead`. It also shows the pattern matching compiled form of these example functions.

7.2 Algorithm for Compiling Pattern-Matching

In this section, an algorithm for compiling the pattern-matching syntax of MPL is presented using the `compile` function.

7.2.1 The `compile` Function

`compile` function takes 3 arguments: a list of sequential terms $[u_1, \ldots, u_n]$, a singleton list of list of pattern phrases formed from the body of the function being compiled, and a default term $t_{\texttt{default}}$, which is initially set to giving an error message. The `compile` function for the function f, of Section 7.1, is:

Function defintion with pattern matching	Function definition with compiled patterns

`append :: [A],[A] -> [A] =`
`  [],   ys   -> ys`
`  x:xs, ys   -> x:append(xs,ys)`
	`append :: [A],[A] -> [A] =`
	`  xl,yl ->`
	`  case xl of`
	`  Nil -> yl`
	`  Cons(x,xs) -> Cons(x,append(xs,yl))`

`pairFun::([Int],[Int]),(Int,Int) ->`
`  (Int,Int) =`
`  ([]  ,[]),(p,q)   -> (p,q)`
`  (x:xs,[]),(p,q)   -> (x,q)`
`  ([],y:ys),(p,q)   -> (p,y)`
`  (x:xs,y:ys),(p,q) -> (x,y)`
	`pairFun::([Int],[Int]),(Int,Int) ->`
	`  (Int,Int) =`
	`  u,v ->`
	`  case Pi_0(u) of`
	`  Nil ->`
	`  case Pi_1(u) of`
	`  Nil  -> (Pi_0(v),Pi1(v))`
	`  Cons(y,ys) -> (Pi_0(v),y)`
	`  Cons(x,xs) ->`
	`  case yls of`
	`  Nil  -> (x,Pi_1(v))`
	`  Cons(y,ys) -> (x,y)`

`codata C -> InfList(A)`
`  = Head :: C -> A`
`  Tail :: C -> C`
 	
`data SF(A) -> C = FF ::   -> C`
`  SS :: A -> C`
 	
`fstOfHead::InfList([A]) -> SF(A) =`
`  (Head := []  ,Tail := t ) -> FF`
`  (Head := a:as,Tail := t ) -> SS(a)`
	`codata C -> InfList(A)`
	`  = Head :: C -> A`
	`  Tail :: C -> C`
	
	`data SF(A) -> C = FF ::   -> C`
	`  SS :: A -> C`
	
	`fstOfHead :: InfList([A]) -> SF(A) =`
	`  u -> case Head(u) of`
	`  Nil -> FF`
	`  Cons(x,xs) -> SF(x)`

Table 7.1: Examples of Pattern Matching Compilation

$$\texttt{compile}\ \ [u_1, \ldots, u_n]$$

$$\left[\left[\begin{array}{lcl} p_{1,1}, \ldots, p_{1,n} & \rightarrow & t_1 \\ & \ldots & \\ p_{m,1}, \ldots, p_{m,n} & \rightarrow & t_m \end{array}\right]\right]\ t_{\texttt{default}}$$

where the first argument is a list of terms, which are variables to begin with. Note that the number of these variables is the same as the number of patterns in any pattern phrase from the singleton list of list of pattern phrases.

7.2.2 Termination Condition and Output

The termination configuration for the `compile` function and the corresponding outputs are:

- Every pattern list in the singleton list of list of pattern phrases is empty, represented as:

$$\texttt{compile}\ \ [\,]\ \ [[\,]\ \ \rightarrow\ t_1, \ldots, [\,]\ \ \rightarrow\ t_m\,]]\ \ t_{\texttt{default}}\ \ :=\ \ t_1$$

- The list of list of pattern phrases is empty, represented as:

$$\texttt{compile}\ \ [\,]\ \ [\,[\,]\,]\ \ t_{\texttt{default}}\ \ :=\ \ t_{\texttt{default}}$$

7.2.3 Execution Steps of `compile` Function

To execute the `compile` function shown below:

$$\texttt{compile}\ \ [u_1, \ldots, u_n]$$

$$\left[\left[\begin{array}{lcl} p_{1,1}, \ldots, p_{1,n} & \rightarrow & t_1 \\ & \ldots & \\ p_{m,1}, \ldots, p_{m,n} & \rightarrow & t_m \end{array}\right]\right] \; t_{\texttt{default}}$$

the algorithm proceeds by looking at the first patterns of every pattern phrase of a singleton list of list of pattern phrases, represented above as $p_{1,1}, \ldots, p_{m,1}$. The algorithm recursively examines the first patterns till a termination condition is achieved. The first patterns $p_{1,1}, \ldots, p_{m,1}$ can either be all variable patterns, all record patterns, all product patterns, all constructor patterns, or a mixture of various patterns. The execution steps for the various cases are described below.

Variable Patterns

If every pattern, $p_{1,1}, \ldots, p_{m,1}$, is a variable pattern, say $v_1, \ldots, v_n$ respectively, then the new configuration of the `compile` function becomes:

$$\texttt{compile} \;\; [u_2, \ldots, u_n]$$

$$\left[\left[\begin{array}{lcl} p_{1,2}, \; \ldots, p_{1,n} & \rightarrow & t_1 \; [u_1/v_1] \\ \vdots & & \\ p_{m,2}, \; \ldots, p_{m,n} & \rightarrow & t_m \; [u_1/v_m] \end{array}\right]\right] \; t_{\texttt{default}}$$

where $t_i \; [u_1/v_i]$, for $1 \leq i \leq m$, means that in the body of the term t_i, the variable term v_i is "substituted" by the term u_1. In general, this is a straight substitution but when u_i is a higher-order destructor, as we shall discuss below, the substitution is more complex.

Record Patterns

If every first pattern, $p_{1,1}, \ldots, p_{m,1}$, is a record pattern of say k destructors $D_1, \ldots, D_k$, where any $p_{i,1}$, for $1 \leq i \leq m$, is of the form:

$$(D_1 \ := \ p_{i,1,1}, \ldots, D_k \ := \ p_{i,1,k})$$

then the new configuration of the `compile` function can be represented as:

$$\texttt{compile} \ ([D_1(u_1), \ldots, D_k(u_1)] \ +\!\!+ \ [u_2, \ldots, u_n])$$

$$\left[\left[\begin{array}{lcl} p_{1,1,1}, \ldots, p_{1,1,k}, p_{1,2}, \ldots, p_{1,n} & \rightarrow & t_1 \\ \vdots & & \\ p_{m,1_1}, \ldots, p_{m,1,k}, p_{m,2}, \ldots, p_{m,n} & \rightarrow & t_m \end{array}\right]\right] \ t_{\texttt{default}}$$

where $D_j(u_1)$, for $1 \leq j \leq k$, represents the destruction of the term u_1 with the j^{th} destructor.

Let us look at an example of compilation of a higher-order record pattern using the above rule. Consider the function `someFun` shown below:

```
codata C -> Exp(A,B) = App :: C,A -> B


someFun :: Exp(A,B),A -> B =
  (App := f),x -> f(x)
```

When pattern-matching on a higher order record, the pattern assigned to a higher-order destructor can only be a variable: trying to look under the variable will result in an error. The `compile` function for `someFun` is:

$$\texttt{compile} \ [\texttt{u1,u2}]$$

$$\left[\left[\texttt{[(App := f),x]} \rightarrow \texttt{f(x)}\right]\right] \ t_{\texttt{default}}$$

Applying the rule for record patterns, one gets:

$$\texttt{compile}\ [\texttt{App(u1),u2)}]\ \left[\left[\left[\texttt{[f,x]} \rightarrow \texttt{f(x)}\right]\right]\right]\ t_{\texttt{default}}$$

Applying the rule for variable patterns one gets:

$$\texttt{compile}\ [\texttt{u2}]\ \left[\left[\left[\texttt{[x]} \rightarrow \texttt{App(u1,x)}\right]\right]\right]\ t_{\texttt{default}}$$

Note that substitution [App(u1)/f] in the body of the term f(x) results in the term App(u1,x) because App is a higher-order destructor. Finally, applying the rule for variable pattern one gets:

$$\texttt{compile}\ []\ \left[\left[\left[\texttt{[ ]} \rightarrow \texttt{App(u1,u2)}\right]\right]\right]\ t_{\texttt{default}}$$

which outputs the term App(u1,u2). The pattern-matching compiled version of function someFun is:

```
someFun :: Exp(A,B),A -> B =
  u1,u2 -> App(u1,u2)
```

Product Patterns

If every first pattern, $p_{1,1}, \ldots, p_{m,1}$, is a product pattern of say k patterns where any $p_{i,1}$, for $1 \leq i \leq m$, is of the form:

$$(p_{i,1,1}, \ldots, p_{i,1,k})$$

then the new configuration of the `compile` function can be represented as:

$$\texttt{compile}\ \ ([\pi_0(u_1), \ldots, \pi_{k-1}(u_1)] \mathbin{+\!\!+} [u_2, \ldots, u_n])$$

$$\left[\left[\begin{array}{lcl} p_{1,1,1}, \ldots, p_{1,1,k}, p_{1,2}, \ldots, p_{1,n} & \rightarrow & t_1 \\ \vdots & & \\ p_{m,1_1}, \ldots, p_{m,1_k}, p_{m,2}, \ldots, p_{m,n} & \rightarrow & t_m \end{array}\right]\right]\ t_{\texttt{default}}$$

where $\pi_{j-1}(u_1)$, for $1 \leq j \leq k$, represents the j^{th} projection of the term u_1.

Constructor Patterns

If every first pattern, $p_{1,1}, \ldots, p_{m,1}$, is a constructor pattern of constructors of a data type D then this execution step is performed.

Suppose D has k constructors such that the i^{th} constructor C_i, for $1 \leq i \leq k$, take s_i number of arguments. Suppose the constructor C_i appears as a constructor pattern x_i number of times in $p_{1,1}, \ldots, p_{m,1}$. These x_i pattern phrases, which start with the constructor C_i as their first pattern phrase, are grouped together in a list. These lists of pattern phrases are then rearranged based on their first constructor patterns such that they appear in the same order in which their corresponding constructors appear in the data definition D. Note that while rearranging the pattern phrases, the relative ordering of pattern phrases having the same first constructors should not be changed. For a constructor which is not present in the list of patterns $p_{1,1}, \ldots, p_{m,1}$, a singleton list is formed with the pattern phrase consisting of that constructor and the default term which is then inserted at the appropriate place

based on its constructor.

The q^{th} pattern in the r^{th} pattern phrase for the group of pattern phrases corresponding to constructor C_i as the first pattern is represented as:

$$p_{r,i,q} \qquad \{1 \leq r \leq x_i \ , \ 1 \leq i \leq k \ , \ 1 \leq q \leq n\}$$

The term corresponding to the r^{th} pattern phrase for the group of pattern phrases corresponding to constructor C_i is represented as:

$$t_{r,i} \qquad \{1 \leq r \leq x_i \ , \ 1 \leq i\}$$

The first pattern for the j^{th} pattern phrase, selected from the list of pattern phrases corresponding to the constructor C_i as the first pattern, is represented as:

$$C_i \ (p_{j,i,1,1}, \ldots, p_{j,i,s_i,1}) \qquad \{1 \leq j \leq x_i \ , \ 1 \leq i \leq k\}$$

As a result of the above steps, the compile function for f will internally look like:

$$\texttt{compile} \ [u_1, \ldots, u_n]$$

$$\left[\left[\begin{array}{lcl} C_1(p_{1,1,1,1}, \ldots, p_{1,1,s_1,1}), \ldots, p_{1,1,n} & \rightarrow & t_{1,1} \\ \vdots & & \\ C_1(p_{x_1,1,1,1}, \ldots, p_{x_1,1,s_1,1}), \ldots, p_{x_1,1,n} & \rightarrow & t_{x_1,1} \end{array}\right], \right.$$

$$\vdots$$

$$\left.\left[\begin{array}{lcl} C_k(p_{1,k,1,1}, \ldots, p_{1,k,s_k,1}), \ldots, p_{1,k,n} & \rightarrow & t_{1,k} \\ \vdots & & \\ C_k(p_{x_k,k,1,1}, \ldots, p_{x_k,k,s_k,1}), \ldots, p_{x_k,k,n} & \rightarrow & t_{x_k,k} \end{array}\right]\right] \ t_{\texttt{default}}$$

which executes to:

$$
\begin{array}{l}
\mathsf{case}\ u_1\ \mathsf{of} \\
\quad C_1(v_1, \ldots, v_{s_1}) \quad \rightarrow \\
\quad\quad \texttt{compile}\ \left([v_1, \ldots, v_{s_1}] +\!\!+ [u_2, \ldots, u_n]\right) \\
\quad\quad\quad \left[\left[\begin{array}{lll} p_{1,1,1,1}, \ldots, p_{1,1,s_1,1}, p_{1,1,2} \ldots, p_{1,1,n} & \rightarrow & t_{1,1} \\ \vdots & & \\ p_{x_1,1,1,1}, \ldots, p_{x_1,1,s_1,1}, p_{x_1,1,2} \ldots, p_{x_1,1,n} & \rightarrow & t_{x_1,1} \end{array}\right]\right]\ t_{\texttt{default}} \\
\quad \vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \\
\quad C_k(v_1, \ldots, v_{s_k}) \quad \rightarrow \\
\quad\quad \texttt{compile}\ \left([v_1, \ldots, v_{s_k}] +\!\!+ [u_2, \ldots, u_n]\right) \\
\quad\quad\quad \left[\left[\begin{array}{lll} p_{1,k,1,1}, \ldots, p_{1,k,s_k,1}, p_{1,k,2} \ldots, p_{1,k,n} & \rightarrow & t_{1,k} \\ \vdots & & \\ p_{x_k,k,1,1}, \ldots, p_{x_k,k,s_k,1}, p_{x_k,k,2}\ \ldots, p_{x_k,k,n} & \rightarrow & t_{x_k,k} \end{array}\right]\right]\ t_{\texttt{default}}
\end{array}
$$

Mixed Patterns

If the first patterns, $p_{1,1}, \ldots, p_{m,1}$, are a mixture of various kinds of patterns then this step is performed. In this step, the similar patterns are grouped together. For example, consider the below singleton list of list of pattern phrases in a compile function where the first pattern phrase has a variable pattern as its first pattern, next two phrases have constructor patterns `Nil` and `Cons` as their first pattern, and the last phrase again has a variable pattern as its first pattern.

$$\texttt{compile}\ [u_1]$$

$$\left[\left[\begin{array}{ll} [\texttt{w}] & \rightarrow\ 1 \\ [\texttt{Nil}] & \rightarrow\ 0 \\ [\texttt{Cons(p,q)}] & \rightarrow\ 2 \\ [\texttt{x}] & \rightarrow\ 1 \end{array}\right]\right]\ t_{\texttt{default}}$$

The similar kinds of patterns are grouped together as shown below:

$$\texttt{compile}\ [u_1] \left[\begin{array}{l} [\texttt{w}]\ \rightarrow\ 1 \\ \left[\begin{array}{ll} [\texttt{Nil}] & \rightarrow\ 0 \\ [\texttt{Cons(p,q)}] & \rightarrow\ 2 \end{array}\right] \\ [\texttt{x}]\ \rightarrow\ 1 \end{array}\right]\ t_{\texttt{default}}$$

One starts with the compilation of the lower most partition of pattern phrases, which then becomes the default term for the second last partition. One then compiles the second last partition which becomes the default term for the partition above it. One continues this process till they reach the first partition.

Applying the rule for mixture patterns on the `compile` function with four pattern phrases shown above, one gets the following:

$$\texttt{compile}\ [\texttt{u1}]\ \left[\left[\ [\texttt{w}]\ \rightarrow\ 1\ \right]\right]$$
$$\left(\texttt{compile}\ [\texttt{u1}]\ \left[\left[\begin{array}{ll} [\texttt{Nil}] & \rightarrow\ 0 \\ [\texttt{Cons(p,q)}] & \rightarrow\ 2 \end{array}\right]\right]\right.$$
$$\left.\left(\texttt{compile}\ [\texttt{u1}]\ \left[\left[\ [\texttt{x}]\ \rightarrow\ 1\ \right]\right]\ t_{\texttt{default}}\right)\right)$$

Chapter 8

λ-Lifting

Once pattern-matching compilation is completed, the next step in the interpretation of MPL programs is λ-lifting. MPL allows the programmers to define local functions. However, Core MPL doesn't allow for local function definitions. Thus all the local MPL functions must be put in the global scope. λ-lifting is the transformation that puts the local functions in the global scope. The defn and where are the two MPL constructs that allow for local function definitions.

In this chapter, the difference between the defn and where constructs, and the strategy for λ-lifting of each contruct is discussed. An algorithm for λ-lifting was given by Thomas Johnsson [6]. The algorithm described in this chapter is an adaptation of Johnsson's algorithm customised for MPL, in particular, bound variables had to be taken into account.

8.1 Local Function Definitions in MPL

Both defn and where can be used to define local functions, however the defn construct doesn't allow non-local variables in the function definitions while the where construct does. Examples of this were described in Section 8.1.1 and Section 8.1.2.

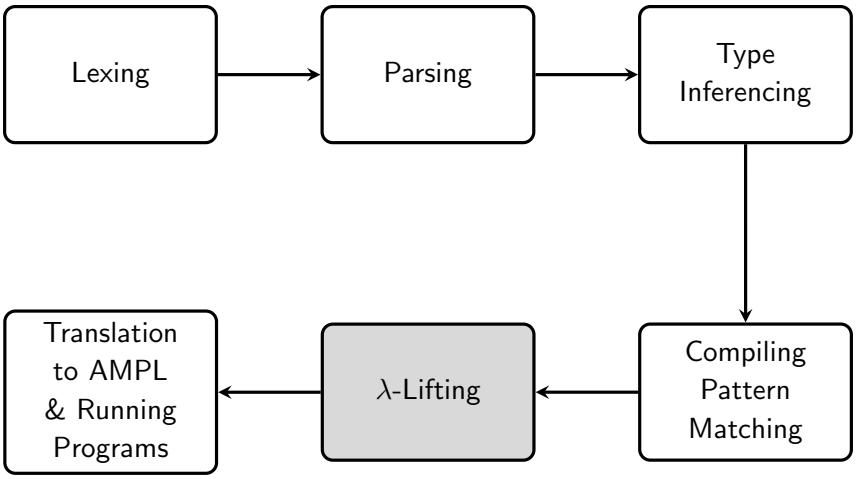

Figure 8.1: Interpretation Stages of MPL

8.1.1 The Defn Construct

The defn construct is used to modularize programs in MPL. The defn construct has the form:

```
defn
  definition_1
      ⋮
  definition_m
where
  definition_1
      ⋮
  definition_n
```

The defn construct has two components:

- **The Header of the Defn Construct**: MPL definitions between the keywords `defn` and `where` form the header of the defn construct. These definitions can be (co)data,

(co)protocol, process, or function definitions and are visible to the entire MPL program.

- **The Body of the Defn Construct**: The body of the defn construct consists of a sequence of definitions. However, these definitions are visible only inside the body and the header of the defn construct. For a (co)data type defined in the body, the (co)data type itself can't appear in the type of a function in the header but the constructors/destructors can be used in the function body.

Table 8.1 shows an example of a pattern-matching compiled MPL program that uses the defn construct. The program defines a `Tree` data structure and a `treeToBST` function. `treeToBST` takes a normal tree and returns a binary search tree (BST) corresponding to the original binary tree. A BST is a binary tree with the following properties: the elements in the left subtree of every node are smaller than the node, the elements in the right subtree of every node are greater than the node element, and duplicate elements are not allowed.

The strategy used in the example in Table 8.1 to convert the binary tree to a binary search tree (BST) makes use of a `List` data structure which is defined in the body of the defn construct. Program extracts the list of integers from the original tree. This list is then used to create a BST by inserting each element of the list into the BST.

The function `treetoBST` defined in the header of the defn construct takes a binary tree and returns a BST by calling the `createBST` function defined in the body of the defn construct.

The constructors of the `List` data type are visible only to the functions in the body and header of the `defn` construct. However, the functions in the header of the defn construct can't include the `List` type in their types and only the functions in the body of the defn can. The functions `append, insertBST, fromListToBST, getIntList,` and `createBST` are visible only to the other functions in the body or to the defintions in the header of the `defn` construct.

This scoping scheme has the advantage that the function `treetoBST` doesn't need to know the underlying implementation of the `createBST` function which can be changed at anytime without impacting any other aspect of the MPL program.

Section 8.2 discusses the λ-lifting algorithm for the defn construct.

8.1.2 The Where Clause

The `where` clause is a sequential MPL construct used to define local functions; where clause are attached to terms. The functions defined are only visible to the other functions of that where clause or within the term to which the where clause is attached.

Table 8.2 shows an example of a program that uses a where clause. The function `exFun` takes two numbers and a list of numbers as input. If the list of numbers is empty, the function outputs `n1`. If the list of numbers are non-empty, the function outputs a sum where every even number in the list contributes `n1` to the sum and every odd number in the list contributes `n2` to the sum.

Function `exFun` has three local functions `ans`, `helpFun1` and `helpFun2` defined using the where clause. Function `ans` doesn't take any arguments. An important thing to notice about functions `helpFun1` and `helpFun2` is that they have *free variables*. The free variables of a function are the set of variables used in the function body that are not the elements of the set of parameters of that function rather they are defined in a higher scope. Free Variables for `helpFun1` and `helpFun2` are { `n1` } and { `n1,n2` } respectively.

Section 8.3 deals with the λ-lifting of local function definitions defined with the `where` construct. The λ-lifting algorithm for the `where` construct is different from that of the `defn` construct because local functions defined using `where` can have free variables.

8.2 λ-Lifting for the Defn Construct

A naive approach to λ-lifting the local functions of `defn` construct would be to directly put all the definitions in the `where` clause of the `defn` construct and the definitions in main body of the `defn` construct, in the global scope. However, there is a problem with this approach as the names of the function definitions in the `where` clause of the `defn` construct may be the

```
defn
  data Tree(A) -> C = Leaf :: -> C
                      Node :: A,C,C -> C

  TreeToBST :: Tree(Int) -> Tree(Int) =
    tree -> createBST(tree)

where
  data List(A) -> C = Nil  ::     -> C
                      Cons :: A,C -> C
  -- append two lists
  append :: List(A),List(A) -> List(A) =
    t1,t2 -> fold t1 of
               Nil  :    -> t2
               Cons :x,r -> Cons(x,r)

  -- take an integer and inserts it into a Binary Search Tree
  insertBST :: Int,Tree(Int) -> Tree(Int) =
    elem,bst ->
      case bst of
        Leaf ->
          Node (elem,Leaf,Leaf)
        Node(n,t1,t2) ->
          case (n == elem) of
            True  -> bst
            False -> case (elem < n) of
                       True  -> Node (n,insertBST (elem,t1),t2)
                       False -> Node (n,t1,insertBST(elem,t2))

  -- takes a list of integers and creates a binary
  -- search tree out of it
  fromListToBST :: List(Int) -> Tree(Int) =
    list -> fold list of
              Nil  :     -> Leaf
              Cons : x,r -> insertBST(x,r)

  -- get a list of integers from a tree of integers
  getIntList :: Tree(Int) -> List(Int) =
    tree -> fold tree of
              Leaf : -> Nil
              Node :x,ls1,ls2 -> append (Cons(x,ls1),ls2)

  -- take a simple binary tree and return a BST
  createBST :: Tree(Int) -> Tree(Int) =
    tree -> fromListToBST (getIntList (tree))
```

Table 8.1: Example : Local functions with defn Construct

```
exFun =
    n1,n2,list -> ans()
       where
          ans =
              -> helperFun1(list)

          helpFun1 =
            ls -> case ls of
                    Nil -> n1
                    Cons(x,xs) -> helperFun2(ls)

          helperFun2  =
            ls -> case ls of
                    Nil ->
                    Cons(y,ys) ->
                      if mod(y,2) == 0
                        then (n1 + helperFun2(ys))
                        else (n2 + helperFun2(ys))
```

Table 8.2: Example : Local Functions with where Construct

same as the names of the function definitions in the global scope. Thus, before the function definitions of the `where` clause of the `defn` construct are lifted to global scope, the functions must be renamed to an unique new global name.

Steps in λ-lifting the local functions of defn construct are listed below:

- Rename the function names of the function definitions in the `where` clause to unique new global names. Once functions are renamed, the function calls made with old names should be altered to reflect the name change.
- The definitions in the main body and the `where` clause of defn can now be lifted to the outermost scope.

The steps have been applied to function definitions in the body of the `defn` construct in the program in Table 8.1. The steps of λ-lifting have been shown in Table 8.3 and 8.4. Table 8.3 shows the step where the local functions in the body of the `defn` construct are renamed to unique global names. Table 8.4 shows the step where the renamed functions in the body

of the `defn` construct are lifted to the outermost scope.

8.3 λ-Lifting for the Where Clause

If one tries lifting the local functions of the `where`, the free variables in the local functions will no longer be in scope. The λ-lifting algorithm for the `where` construct is thus more complicated than that for the `defn` construct because of the presence of non-local variables in the local function definitions.

The λ-lifting algorithm for the local functions of the where construct consists of the following steps:

1. **Rename functions and their parameters**: Functions in different scopes can have the same name, however, once the λ-lifting transformation is performed, all the functions will be in the same scope and thus they need different names. Thus, the local functions must be renamed. Care must be taken to convert the function calls made with old function names to function calls with new function names.

   The arguments of all the function definitions are also renamed with fresh variables to ensure that every function has unique parameters. This step is required for the third step of the algorithm.

2. **The Set Equations for functions**: For every local function f defined in the `where` construct, a triple is computed.

   $$(fv_f, bv_f, fl_f)$$

   where fv_f is the set of free variables, bv_f is the set of bound variables, and fl_f is the set of functions called in the function f.

   The pair of a function name and the triple for that function is called a *Set Equation*

```
defn
  data Tree(A) -> C = Leaf :: -> C
                      Node :: A,C,C -> C

  TreeToBST :: Tree(Int) -> Tree(Int) =
    tree -> unqFun5(tree)

where -- this is the where part of the defn construct
  data List(A) -> C = Nil  ::     -> C
                      Cons :: A,C -> C

  -- append two lists
  unqFun1 :: List(A),List(A) -> List(A) =
    t1,t2 -> fold t1 of
               Nil  :    -> t2
               Cons :x,r -> Cons(x,r)

  -- take an integer and inserts it into a Binary Search Tree
  unqFun2 :: Int,Tree(Int) -> Tree(Int) =
    elem,bst ->
      case bst of
        Leaf -> Node(elem,Leaf,Leaf)
        Node(n,t1,t2) ->
          case (n == elem) of
            True  -> bst
            False -> case (elem < n) of
                       True  -> Node(n,unqFun2 (elem,t1),t2)
                       False -> Node(n,t1,unqFun2 (elem,t2))

  -- takes a list of integers and creates a binary
  -- search tree out of it
  unqFun3 :: List(Int) -> Tree(Int) =
    list -> fold list of
              Nil  :     -> Leaf
              Cons : x,r -> unqFun2 (x,r)

  -- get a list of integers from a tree of integers
  unqFun4 :: Tree(Int) -> List(Int) =
    tree -> fold tree of
              Leaf : -> Nil
              Node :x,ls1,ls2 -> unqFun1 (Cons(x,ls1),ls2)

  -- take a simple binary tree and return a BST
  unqFun5 :: Tree(Int) -> Tree(Int) =
    tree -> unqFun3 (unqFun4 (tree))
```

Table 8.3: Step 1 : Rename local functions of the defn construct

```
data List(A) -> C = Nil  ::     -> C
                  Cons :: A,C -> C

data Tree(A) -> C = Leaf :: -> C
                  Node :: A,C,C -> C

-- append two lists
unqFun1 :: List(A),List(A) -> List(A) =
  t1,t2 -> fold t1 of
             Nil  :    -> t2
             Cons :x,r -> Cons(x,r)

-- take an integer and inserts it into a Binary Search Tree
unqFun2 :: Int,Tree(Int) -> Tree(Int) =
  elem,bst ->
    case bst of
      Leaf -> Node(elem,Leaf,Leaf)
      Node(n,t1,t2) ->
        case (n == elem) of
          True  -> bst
          False -> case (elem < n) of
                     True  -> Node(n,unqFun2 (elem,t1),t2)
                     False -> Node(n,t1,unqFun2 (elem,t2))

-- takes a list of integers and creates a binary
-- search tree out of it
unqFun3 :: List(Int) -> Tree(Int) =
  list -> fold list of
            Nil  :     -> Leaf
            Cons : x,r -> unqFun2 (x,r)

-- get a list of integers from a tree of integers
unqFun4 :: Tree(Int) -> List(Int) =
  tree -> fold tree of
            Leaf : -> Nil
            Node :x,ls1,ls2 -> unqFun1 (Cons(x,ls1),ls2)

-- take a simple binary tree and return a BST
unqFun5 :: Tree(Int) -> Tree(Int) =
  tree -> unqFun3 (unqFun4 (tree))

TreeToBST :: Tree(Int) -> Tree(Int) =
  tree -> unqFun5(tree)
```

Table 8.4: Step 2 : Lift the definitions of defn to the global scope

of the function:

$$(f, (fv_f, bv_f, fl_f))$$

A set of set equations are obtained as a result of this step.

3. **Solve the Set Equations using a fixed point calculation**: The aim of this step is to generate the set of all the free variables needed by a function f. This set not only contains the free variables that are directly present in the body of the function f but also the set of free variables present in the body of functions called inside f.

   Once the set of set equations corresponding to the local functions defined in the `where` clause are generated, the equations are then solved to get the all the free variables for these functions. The set equations are solved using a fixed point calculation.

   The fixed point of a function p is a value x, such that $p\ \ x\ =\ x$.

   An algorithm for solving the set equations is shown in Table 8.5. A set equation achieves a fixed point condition when the set of free variables does not change from one iteration to another.

4. **Add the free variables to their corresponding functions**: Once all the free variables corresponding to a function are generated, they are added to the parameters of the function definition. This ensures that there are no free variables in the function body. Since the arity of the function has changed, the arguments of the function call must be expanded with the free variables of that function.

5. **Lift the local functions to global scope**: Since there are no free variables in the local functions any more, they can be lifted into the global scope. This is the final step in the λ-lifting transformation.

The stepwise λ-lifting transformation for the local function in the `where` construct of function `exFun` defined in Table 8.2 is shown in Tables 8.6 and 8.7.

SolveSetEqns

input: A set of type equations es

output: A set of type equations that have achieved fixpoint

for each $e \in es$

$(f, (fv, bv, fl)) \leftarrow e$

repeat

for every $g \in fl$

$(fv_1, bv_1, fl_1) \leftarrow$ lookup g in $(es \setminus e)$

$fv' = (fv \cup fv_1) \setminus bv_1$

$fl' = (fl \cup fl_1) \setminus g$

$e' = (f, (fv', bv_1, fl'))$

until $fv == fv'$ {This is the fixed point condition}

return e'

Table 8.5: Algorithm for Solving Set Equations

Original Program

```
exFun =
  n1,n2,list -> ans()
    where
      ans =
             -> helperFun1 (list)
      helperFun1 =
          ls -> case ls of
                  Nil -> n1
                  Cons(x,xs) -> helperFun2(ls)
      helperFun2 =
          ls ->
            case ls of
              Nil -> 0
              Cons(y,ys) ->
                if mod(y,2) == 0
                  then n1 + helperFun2(ys)
                  else n2 + helperFun2(ys)
```

Step 1 : Renamed function names and parameters

```
exFun =
  n1,n2,list -> unq_fn1()
    where
      unq_fn1 =
          -> unq_fn2(list)
      unq_fn2 =
        u1 -> case ls of
                Nil -> n1
                Cons(x,xs) -> unq_fn3(u1)
      unq_fn3 =
        u2 ->
          case ls of
            Nil -> 0
            Cons(u3,u4) ->
              if mod(u3,2) == 0
                then n1 + unq_fn3(u4)
                else n2 + unq_fn3(u4)
```

Step 2: Generate Set Equations for Local Functions

```
(unq_fun1,({list},{},{unq_fun2}))
(unq_fun2,({n1},{u1},{unq_fun3}))
(unq_fun3,({n1,n2},{u2},{unq_fun3}))
```

Table 8.6: λ-Lifting exFun (Continued on Table 8.7)

Step 3: Solve Set Equations

The solved set equations are as follows:

```
(unq_fun1,({list,n1,n2},{},{unq_fun3}))
(unq_fun2,({n1,n2},({u1},{unq_fun3}))
(unq_fun3,({n1,n2},({u2},{unq_fun3}))
```

Thus, the free variables for `unq_fun1` are {`list,n1,n2` }. For functions `unq_fun2` and `unq_fun3`, the free variables are {`n1,n2` }.

Step 4: Augment free variables to function parameters and call

```
exFun =
  n1,n2,list ->
      helperFun1 (list,n1,n2)
    where
      unq_fn1 =
        u1 -> case ls of
                Nil -> n1
                Cons(x,xs) -> unq_fn2(u1,n1,n2)
      unq_fn2 =
        u2 ->
          case ls of
            Nil -> 0
            Cons(u3,u4) ->
              if mod(u3,2) == 0
                then n1 + unq_fn2(u4,n1,n2)
                else n2 + unq_fn2(u4,n1,n2)
```

Step 5: Lift Local Functions to Outermost Scope

```
unq_fn2 =
  u2 ->
    case ls of
      Nil -> 0
      Cons(u3,u4) ->
        if mod(u3,2) == 0
          then n1 + unq_fn2(u4,n1,n2)
          else n2 + unq_fn2(u4,n1,n2)
unq_fn1 =
  u1 -> case ls of
          Nil -> n1
          Cons(x,xs) -> unq_fn2(u1,n1,n2)
exFun =
  n1,n2,list -> unq_fn1(list,n1,n2)
```

Table 8.7: Lambda Lifting `exFun` (Continued from Table 8.6)

Chapter 9

Abstract Machine for MPL (AMPL)

In the previous chapters the different stages of interpretation of MPL programs (listed in Figure in 9.1) have been described. The last stage of interpretation deals with running MPL programs on MPL's abstract machine (called AMPL). However, before an MPL program can be run on AMPL, it needs to be translated to a list of commands that AMPL can execute. This translated code is called AMPL code.

In this chapter, the translation of MPL to AMPL code and the execution of the AMPL code is discussed.

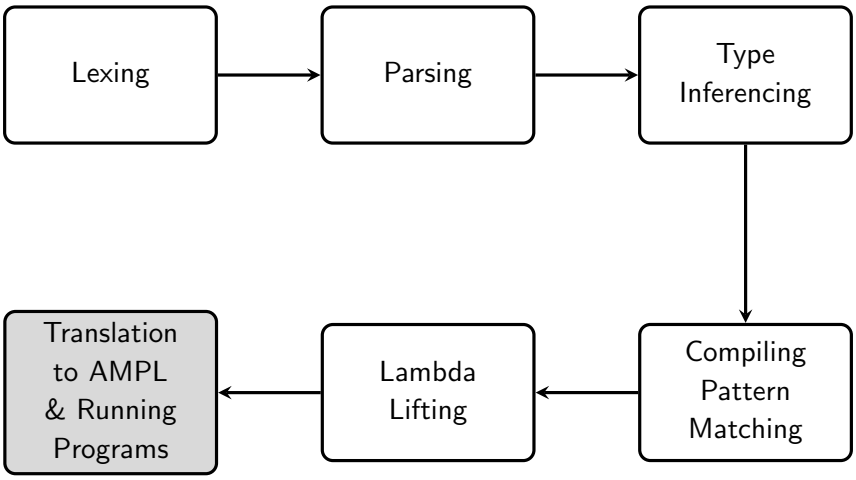

Figure 9.1: Interpretation Stages of MPL

9.1 Introduction to Abstract Machines

Abstract machines are useful conceptual tools when implementing a programming language because they omit the many details of real machines and thus bridge the gap between the programming language and the physical machine. An added advantage of abstract machines is the ease of porting the language to different platforms. An abstract machine reduces the problem of porting the programming language to the problem of porting the abstract machine. This is an easy task as the abstract machines are simpler and smaller than the programming language itself. This strategy was used by the programming language Java which employed Java Virtual Machine (JVM) [17] and Prolog which employed Warren's Abstract Machine (WAM) [18]. Some examples of abstract machines for functional languages are:

- Landin's SECD Machine [7] was one of the first abstract machines for functional programming languages specifically designed to evaluate lambda calculus expressions *by value*.
- Modern-SEC Machine [8] is an improved and efficient version of the SECD machine.
- Cardelli's Functional Abstract Machine (FAM) [9] is an extended and optimized SECD machine. It was used in the first native-code implementation of ML.
- Cousineau, Curien and Mauny's Categorial Abstract Machine (CAM) [12]. Its instructions correspond to the constructions of a *Cartesian Closed Category*: identity, composition, abstraction, application, pairing, and selection. It was the basis for the CAML programming language.
- Three Instruction Machine (TIM) [13] is a simple abstract machine for evaluation of super-combinators.
- Spineless-Tagless G-Machine (STG-Machine) [16] is the abstract machine for Haskell.

9.2 Introduction to AMPL

AMPL is an abstract machine which runs MPL programs. Just as MPL is separated into two levels, sequential MPL and concurrent MPL, so too can AMPL can be thought of as two machine levels:

- Sequential AMPL (SAMPL) which runs sequential MPL code
- Concurrent AMPL (CAMPL) which runs concurrent MPL code

This conceptual separation means AMPL has a modular description and design. Thus, either of the AMPL components can be modified without disturbing the other.

Sequential AMPL had to accomodate both inductive and coinductive data types. To acheive this it used new constructs: constructor, destructor, record, and case. The design of the concurrent AMPL was essentially entirely new. It is notable for its succinct description.

9.3 Sequential Abstract Machine for MPL (SAMPL)

Sequential MPL code compiles to Sequential AMPL or SAMPL code. SAMPL is inspired by modern-SEC machine [8]. However, the modern-SEC machine doesn't have explicit commands to which data types and codata types can be compiled. SAMPL augments the basic modern-SEC machine with case and constructor commands to compile data types and record and destructor commands to compile codata types. SAMPL also adds commands for basic functions like addition, multiplication, equality testing etc which are built-in to the language. The SAMPL commands are evaluated *by-value* except for the record and destructor - the codata type commands - which are lazily evaluated.

Table 9.1 provides a list of SAMPL commands and a brief description of each command.

Instruction	Explanation
Store	pushes the top stack element into the environment
Access(n)	put n^{th} value in the environment onto the stack.
Ret	return the top stack value and jump to the continuation below
call $\langle$code$\rangle$	jump to the code
Built in instructions:	
$\text{Const}_T(k)$	push the constant k of basic type T on the stack
Add	Pop two arguments from the top of the stack and add them
Mul	Pop two arguments from the top of the stack and mlutiply them
Leq	Pop two arguments from the top of the stack and compare them
	etc.
Data instructions:	
$\text{Cons}(i, n)$	push the i^{th} constructor onto the stack with arguments the top n elements of the stack, $\text{Cons}(i, s_1, ..., s_n)$.
$\text{Case}[c_1, ..., c_n]$	when $\text{Cons}(i, t_1, ..., t_n)$ is on the stack remove it and push $t_1, ..., t_n$ into the environment and evaluate c_i.
Codata instructions:	
$\text{Rec}[c_1, ..., c_n]$	create a record on the stack with current environment, $\text{rec}([c_1, ..., c_n], e)$
$\text{Dest}(i, n)$	destruct a record: choose the i^{th} function closure (c_i, e) and run c_i in environment e supplemented with the first n values on the stack.

Table 9.1: SAMPL Commands

9.3.1 Compilation of Sequential MPL to SAMPL Commands

The sequential MPL code is compiled to a list of SAMPL commands which are then executed on the abstract machine described in Table 9.2. The compilation scheme is described below:

$$[\![\mathsf{record}\ \{D_i\ x_{1_i}, ..., x_{n_i} \mapsto t_i\}_{i\in[1,m]}\ ]\!]_v = \mathsf{Rec}[[\![t_i]\!]_{x_1,...,x_{n_i},v}\ \mathsf{Ret}]_{i\in[1,m]} \quad (9.1)$$

$$[\![(D_i(x, t_1, ..., t_n)]\!]_v = [\![t_n]\!]_v ... [\![t_1]\!]_v\ [\![x]\!]_v \mathsf{Dest}(i, n) \quad (9.2)$$

$$[\![C_i(t_1, .., t_n)]\!]_v = [\![t_n]\!]_v ... [\![t_1]\!]_v\ \mathsf{Cons}(i, n) \quad (9.3)$$

$$[\![\mathsf{case}\ t\ \{C_i\ x_{1_i}, ..., x_{n_i} \mapsto t_i\}_{i\in[1,m]}\ ]\!]_v = [\![t]\!]_v \mathsf{Case}[[\![t_i]\!]_{x_1,...,x_{n_i},v}\ \mathsf{Ret}]_{i\in[1,m]} \quad (9.4)$$

$$[\![x]\!]_v = \mathsf{Access}(n) \quad \text{where } n = \mathsf{index}\ v\ x \quad (9.5)$$

$$[\![a\ \text{op}\ b]\!]_v = [\![b]\!]_v\ [\![a]\!]_v \mathsf{Op} \quad (9.6)$$

$$[\![k]\!]_v = \mathsf{Const}_T(k) \quad (9.7)$$

$[\![\ ]\!]_v$ signifies the compilation of a *Core MPL's* sequential constructs to *SAMPL* commands in context of an *environment* v. The environment acts as a repository of variables used in a piece of code being compiled. It is used to replace a variable name with the relative position in the program body. A description of these compilation steps is:

(9.1) is the compilation scheme for the record construct. D_i is the i^{th} destructor and t_i is the corresponding term for that destructor in the record.

(9.2) is the compilation scheme for a destructor represented by D_i. The first argument of D_i is the record to be destructed and the subsequent arguments are the arguments of D_i.

(9.3) is the compilation of a constructor C_i. The terms $t_1, \ldots, t_n$ which are the arguments of the constructors are recursively compiled and concatenated in the order opposite to which they occur. The constructor name is replaced in the compilation process by a pair, the constructor number and the number of arguments that the constructor takes.

(9.4) is the compilation of constants (integers,floats,characters).

(9.5) is the compilation of a case construct. A few things worth noting in the compilation of case constructs are as follows:

- The term t_i of the C_i phrase is compiled in the context enhanced with the arguments of C_i.
- Ret (return) should be the last command in the compiled code corresponding to any constructor.
- Before the compilation of the case construct starts, it should be ensured that the constructors of the data type are arranged in the order in which they are defined.

(9.6) is the compilation of a variable x. The variable is looked up in the context v. the depth of the variable in the context is the argument to the Access command.

(9.7) is the compilation of infix functions. These functions are converted to their postfix forms with their arguments recursively compiled.

9.3.2 Transition Table for SAMPL

Once Sequential MPL is compiled to SAMPL code, the code can then be executed on the abstract machine described in Table 9.2. The *Code* is represented as a list of SAMPL commands. Two other structures namely the *Environment* and the *Stack* need to be introduced in order to show how the machine works. Both *Environment* and *Stack* are Last In First Out (LIFO) data structures. In our prototype implementation of MPL, they are implemented using Haskell lists.

The *Environment* keeps track of which variables are visible in the code at a given point and is thus a transient data structure: it changes as one traverses through the code. The *Stack* acts as storage for the intermediate values and the final result during the execution of the code.

Before			After		
Code	Env	Stack	Code	Env	Stack
$\mathsf{Store}; c$	e	$v : s$	c	$v : e$	s
$\mathsf{Access}(n); c$	e	s	c	e	$e(n) : s$
$\mathsf{Call}(c) : c'$	e	s	c	e	$\mathsf{clos}(c', e) : s$
$\mathsf{Ret} : c$	e	$v : \mathsf{clos}(c', e') : s$	c'	e'	$v : s$
$\mathsf{Cons}(i, n) : c$	e	$v_1 : ..., v_n : s$	c	e	$\mathsf{cons}(i, [v_1, .., v_n]) : s$
$\mathsf{Case}(c_1, ..., c_n) : c$	e	$\mathsf{Cons}(i, [v_1, ..., v_n]) : s$	c_i	$v_1 : .. : v_n : e$	$\mathsf{clo}(c, e) : s$
$\mathsf{Rec}(c_1, .., c_n) : c$	e	s	c	e	$\mathsf{rec}([c_1, ..., c_n], e) : s$
$\mathsf{Dest}(i, n) : c$	e	$\mathsf{rec}([c_1, .., c_n], e') : v_n : .. : v_1 : s$	c_i	$v_1 : .. : v_n : e'$	$\mathsf{clo}(c, e) : s$
$\mathsf{Const}_T(k) : c$	e	s	c	e	$\mathsf{const}_T(k) : s$
$\mathsf{Add} : c$	e	$n : m : s$	c	e	$(n + m) : s$
$\mathsf{Mul} : c$	e	$n : m : s$	c	e	$(n * m) : s$
$\mathsf{Leq} : c$	e	$n : m : s$	c	e	$(n \leq m) : s$

Table 9.2: Machine Transitions for the SAMPL

The state of SAMPL at any given time can be described as the triple of the compiled SAMPL *Code*, *Environment*, and *Stack* represented as (C, E, S). Executing SAMPL code means starting with an initial state and changing states based on the transition table for SAMPL in Table 9.2 until a final state is reached. The final state contains the result of the computation. The selection of transition at any step of code execution is done by pattern matching the machine state against the transition table. Thus, the given description of the SAMPL is easily implemented in Haskell using its pattern-matching.

The initial state of the machine is formed by putting the compiled code as the first argument of the triple and intialising the *Enviornment* and *Stack* as empty lists.

$$\text{Initial State} : (C,\ [\,],\ [\,])$$

The final state is reached when the *Code* and the *Environment* is empty. The top of the *Stack* contains the output value.

$$\text{Final State} : ([\,],\ [\,],\ S)$$

In Table 9.2, $\mathsf{clos(c,e)}$ denotes closure of *Code* c with *Environment* e and $\mathsf{e(n)}$ is the n^{th}-element of the environment.

MPL Program corresponding to $(\lambda \mathbf{x}.\ \mathbf{x} + \mathbf{1})\ 2$

```
fun f1  =
   ->  App(record(App(y) :=  y + 1),2)
```

Step wise compilation to SAMPL Code

$[\![\mathsf{App}(\mathsf{record}\ (\mathsf{App(y)} := y + 1, 2))]\!]_{[\,]}$ (Dest. rule)

$[\![2]\!]_{[\,]}$ ++ $[\![\mathsf{record}\ (\mathsf{App(y)} := y + 1)]\!]_{[\,]}$ ++ [Dest 1 1] (Const. & Rec rule)

[CInt 2] ++ Rec$\Big[[\![y + 1]\!]$ ++ [Ret]$\Big]$ ++ [Dest 1 1] (Op (Infix) rule)

[CInt 2] ++ Rec$\Big[[\![1]\!]_{[y]}$ ++ $[\![y]\!]_{[y]}$ ++ [Add] ++ [Ret]$\Big]$ ++ [Dest 1 1] (Const. & Var. rule)

[CInt 2] ++ Rec$\Big[$[CInt 1] ++ [Access 1] ++ [Add] ++ [Ret]$\Big]$ ++ [Dest 1 1]

$\Big[$CInt 2, Rec$\big[$CInt 1, Access 1, Add, Ret$\big]$, Dest 1 1$\Big]$ (SAMPL code)

Table 9.3: Example : Compilation of Sequential MPL to SAMPL Code

Code	Env	Stack
CInt(2); Rec[c]; Dest 1 1	ε	ε
Rec[c]; Dest 1 1	ε	cint 2 : ε
Dest 1 1	ε	rec([c], ε) : cint 2 : ε
CInt 1; Access 1; Add; Ret	cint2 : ε	clo(ε, ε) : ε
Access(1) : Add : Ret	cint2 : ε	cint 1 : clo(ε, ε) : ε
Add : Ret	cint 2 : ε	cint 2 : cint 1 : clo(ε, ε) : ε
Ret	cint 2 : ε	cint 3 : Clo(ε, ε) : ε
ε	ε	cint 3 : ε

where `c := [CInt 1,Access 1,Add,Ret]`

Table 9.4: Executing Code on SAMPL

Table 9.3 provides an example of the step by step compilation sequential MPL to SAMPL. The compilation steps are labelled to identify which rules are used in each step. The Sequential MPL code being compiled is a function `f1`. The function `f1` is equivalent to the lambda application $(\lambda x.x + 1)\ 2$. The function `f1` demonstrates MPL's technique of implementing higher-order functions using codata types.

Table 9.4 shows an example of the step by step execution of the SAMPL commands generated for function `f1` in Table 9.3.

Execution of the code starts with empty *Environment* and *Stack*. In the final step *Code* and the *Environment* are empty and the result `CInt 3` is obtained on top of the *Stack*.

9.4 Concurrent Abstract Machine for MPL (CAMPL)

Concurrent Abstract Machine for MPL (CAMPL) is the machine on which the Concurrent MPL code is run. Concurrent MPL code is comprised of processes. The state of a process is represented as a four tuple of *Stack, Translation, Environment, and Code* (S, t, E, C). *Stack, Environment and Code* are also used to represent the state of Sequential MPL programs and have the same meaning here.

- The *Stack* holds the intermediate value and the final result when executing a process.
- The *Translation* is an additonal structure used in describing the state of a *Concurrent*

MPL program. This additonal structure is required because Concurrent MPL programs have channels, a feature that the sequential MPL programs lack. *Translation* acts as a map of the local channel names to global channel names. The channel connecting two processes may be named differently inside each process. Processes use the *Translation* to determine how local channels correspond to gobal channels.

- The *Environment* is used to determine which variables are visible at a given point of the Concurrent program.
- The *Code* is the list of Concurrent AMPL commands corresponding to the MPL program.

The initial state of CAMPL is formed by loading the compiled code corresponding to Concurrent MPL program (processes) as the fourth argument of the tuple, the *Translation* of channels corresponding to a process as the second argument, and intialising the *Environment* and *Stack* which are the first and the third arguments respectively, as empty lists. The initial state is represented as:

$$\text{Initial State} : (\ [\ ], t, [\ ], C)$$

The final state is reached when the *Code*, the *Environment* and the *Translation* is empty. The top of the stack contains the output of the code. The final state is represented as:

$$\text{Final State} : (S, [\ ], [\ ], [\ ])$$

CAMPL consists of two main components, the Channel Mananger $\mathcal{C}$, and the Process Manager $\mathcal{P}$.

9.4.1 The Channel Manager

The **Channel Manager** $\mathcal{C}$ is a set of pairs of channels and their queues. It is represented as $\mathcal{C}\{(\alpha, q' \mid q)\}$ which indicates that a channel α has the communication queue $q' \mid q$ associated

with it. A communication queue has two parts: the output queue q' and the input queue q. Each channel associated to a process is either an input or output polarity channel: the process writes its communication for an input polarity channel onto the input queue of that channel and for an output polarity channel on the output queue. We add a communication, x, to the back of the input queue by writing $q : x$ and we shall also use this as a pattern to indicate that x is the last item on that queue. Similarly on the output queue we shall add a communication item, x to the front of the queue $x : q'$. To access items on queues we shall use pattern-matching: for example, to access the first-in item, x, of the input queue of a channel which is the last item as we are writing it we shall use the pattern $q : x$. We shall write the empty queue as ϵ. Note that it is the two first-in items x and y in a communication queue $x : q'|q : y$ which are subject to communication actions (described in Section 9.4.6).

9.4.2 The Process Mananger

The Process Manager is a set of processes. It is represented as $\mathcal{P}\ \{(S, t, E, C)\}$ signifying that it has selected one process that will be advanced by one execution step. The Process Manager in theory selects this process non-deterministically. In practise, it may be sensible to execute the selected process not just by one step but for mutliple steps, or until it has a concurrent action before interrupting it.

9.4.3 Interaction between Process Manager and Channel Manager

The interaction between the Process Manager and the Channel Manager can be understood with the help of some examples.

Suppose there are two MPL processes P_1 and P_2 connected by a channel α, which acts as an output polarity channel for process P_1 and an input polarity channel for process P_2. Suppose P_1 puts a value on α and P_2 receives that values on α. Since MPL assumes no fixed order of execution of processes, the process receiving the value on the channel (P_2) may be executed before the process putting the value on the channel (P_1). When process

P_2 is executed before P_1, the Channel Manager suspends the process P_2 and attaches it to the input queue of the channel α. When process P_1 is scheduled and executed, the value is obtained on α and put on its output queue. Once Channel Manager sees the configuration where a suspended process is attached to one queue of the channel and a value is present on the other queue of the channel, it reactivates the suspended process and puts it in the set of active processes with the value transmitted. The Process Manager can then schedule this active process for execution.

Here the interaction between the Process Manager and Channel Manager is explained in terms of just the get-put command pair. However, similar interaction between the two components of CAMPL takes places for split-fork, hput-hcase and close-halt pairs.

9.4.4 Concurrent Commands

In order to understand the working of CAMPL, we start by considering the CAMPL commands. Table 9.5 lists the concurrent commands with a brief description of each. The execution of CAMPL commands are determined by the action of the Process Manager which uses the Channel Manager. The Process Manager's actions are described in Section 9.4.5 and the Channel Manager's actions are described in Section 9.4.6.

9.4.5 Process Manager's Actions

This section describes the execution step corresponding to the CAMPL commands. When a command executes it not only changes the state of the running process shown by $\mathcal{P}\{(s, t, e, c)\}$ but may also change the state of the queues associated with a channel. Hence, the transition table for CAMPL (described in Table 9.6) have columns for both Process Manager and Channel Manager in the **before** and the **after** column. An explanation of the Process Manager's actions corresponding to the various CAMPL commands is provided below:

Get/Put: These are the two basic communication commands. put transmits a value on a

get α; C	get a value on channel α
put α; C	put a value on channel α
split α into $(\alpha_1, \alpha_2); C$	split channel α into two (new) channels
fork α as α_1 with $\Gamma_1.C_1$ α_2 with $\Gamma_2.C_2$	forking on a channel into two distinct processes
hput α $n; C$	put a "handle" on channel α
hcase$\{C_1, .., C_n\}$	the cases on receiving a "handle"
close $\alpha; C$	closing a channel
halt α	halting process attached to a single channel
plug $[\alpha_1, ..., \alpha_n]$ $\Gamma_1.C_1$ $\Gamma_2.C_2$	two processes to communicate on n channels
run t $\langle$process$\rangle$	runs a process with local channel to caller channel translation t
id $\alpha = \beta$	identifying channels.

Table 9.5: Basic Concurrent Commands

communication channel and get receives a value on a communication channel. To put a value on an output polarity channel, one simply places the value on the output queue of that channel. To get a value on an output polarity channel, the process suspends itself on the output queue of the channel with the demand for a value. Similarly, for an input polarity channel, get and put place the suspension of the process and the value to be put on the input queue of the channel.

Split: The split instruction splits the channel α into two channels (α_1 and α_2). One therefore adds to the appropriate queue notification of the two "global" channels into which the channels are split. This means one must choose two new channel names (here β_1 and β_2), and must remember the translation from the local channel names, $t[\beta_1/\alpha_1, \beta_2/\alpha_2]$. Once split is done, the machine continues with the execution of the remaining code.

Fork: The Fork command is dual to the split command. The fork command creates two processes which are supposed to communicate on the new channels assigned by a split command. However, the splitting may not have happened when the fork command is executed. Thus, on a fork command the process suspends itself and attaches itself to the channel which is to be split. The process does not fork, however until the

corresponding channel action of splitting is performed (as a communication action) and at that stage the translations for the local channel names to global names are adjusted and the forked processes are enabled (i.e. added into the process manager).

Plug: The plug command allows two processes to communicate along certain channels. As with a fork command, the channels of the process must be divided amongst the two process being plugged together. In addition, new global communication channel names must be assigned to the channels along which the processes wish to communicate. After a plug command both processes are enabled (i.e. added into the process manager).

Handle: The hput command, which sends a "handle" – a protocol constructor – which is matched by a listening process which will react according to which handle is received by an hcase command. These commands behave somewhat like a "put" and "get", except that for the latter one chooses the code which is to be run based on the handle received rather than simply using the value in the subsequent code.

Call: This "jumps" to some code for a predefined process. The only subtlety in the call command is that one must start by setting up the translation of the local channels of the predefined process into global channel names.

Close/Halt: Closing a channel, close α, causes the channel to be removed. Corresponding to closing a channel is the halt command. Only when all other channels of a process are closed can a process call a halt α on the one remaining channel.

Id $\alpha := \beta$: This provides an identity map between two channels. It alters the translation of the process by defining the translation of α to be the same as the translation of β. This command is used, in particular, when "bending" wires to simulate the negation of linear logic.

$\mathcal{C}\{(t(\alpha), q' \mid q)\}$	$\mathcal{P}\{(s,t,e,\mathsf{get}\ \alpha;c)\}$	$\mathcal{C}\{(t(\alpha), q' \mid q{:}\mathsf{g}(s,t,e,c))\}$	$\mathcal{P}\{\}$
$\mathcal{C}\{(t(\alpha), q' \mid q)\}$	$\mathcal{P}\{(v{:}s,t,e,\mathsf{put}\ \alpha;c)\}$	$\mathcal{C}\{(t(\alpha), q' \mid q{:}v)\}$	$\mathcal{P}\{(s,t,e,c)\}$
$\mathcal{C}\{(t(\alpha), q' \mid q)\}$	$\mathcal{P}\{([],t,e,\begin{matrix}\mathsf{split}\ \alpha\ \mathsf{into}\\(\alpha_1,\alpha_2);c\end{matrix})\}$	$\mathcal{C}\left\{\begin{pmatrix}t(\alpha),\\q' \mid q{:}\langle\beta_1,\beta_2\rangle\end{pmatrix}\right\}$	$\mathcal{P}\{([],t\begin{bmatrix}\beta_1/\alpha_1\\\beta_2/\alpha_2\end{bmatrix},e,c)\}$
$\mathcal{C}\{(t(\alpha), q' \mid q)\}$	$\mathcal{P}\{(s,t,e,\mathsf{close}\ \alpha;c)\}$	$\mathcal{C}\{t(\alpha), q' \mid q{:}\mathsf{close})\}$	$\mathcal{P}\{(s,t\backslash\alpha,e,c)\}$
$\mathcal{C}\{(t(\alpha), q' \mid q)\}$	$\begin{matrix}\mathsf{fork}\ \alpha\ \mathsf{as}\\\mathcal{P}\{([],t,e,\alpha_1\ \mathsf{with}\ \Gamma_1.c_1\,,[])\}\\\alpha_2\ \mathsf{with}\ \Gamma_2.c_2\end{matrix}$	$\mathcal{C}\{\begin{pmatrix}t(\alpha),\\q' \mid q{:}\begin{bmatrix}t,e,\alpha_1/\Gamma_1.c_1\\\alpha_2/\Gamma_2.c_2\end{bmatrix}\end{pmatrix}\}$	$\mathcal{P}\{\}$
$\mathcal{C}\{\}$	$\mathcal{P}\{([],t,e,\mathsf{id}\ \alpha:=\gamma;c,[])\}$	$\mathcal{C}\{\}$	$\mathcal{P}\{([],t[t(\gamma)/\alpha],e,c)\}$
$\mathcal{C}\{\}$	$\begin{matrix}\mathsf{plug}[\alpha_1,...,\alpha_n]\\\mathcal{P}\{([],t,e,\ \Gamma_1.c_1\ \ ,[])\}\\\Gamma_2.c_2\end{matrix}$	$\mathcal{C}\{(\gamma_i,\varepsilon \mid \varepsilon)_{i=1...n}\}$	$\mathcal{P}\left\{\begin{matrix}([],t[\gamma_i/\alpha_i]_{\Gamma_1},e,c_1),\\([],t[\gamma_i/\beta_i]_{\Gamma_2},e,c_2)\end{matrix}\right\}$
$\mathcal{C}\{(t(\alpha), q' \mid q)\}$	$\mathcal{P}\{([],t,e,\mathsf{halt}\ \alpha)\}$	$\mathcal{C}\{t(\alpha), q' \mid q{:}\mathsf{halt})\}$	$\mathcal{P}\{\}$
$\mathcal{C}\{\}$	$\mathcal{P}\{(s,t,e,\mathsf{run}\ t'\ c)\}$	$\mathcal{C}\{\}$	$\mathcal{P}\{(s,t;t',e',c)\}$
$\mathcal{C}\{(t(\alpha), q' \mid q)\}$	$\mathcal{P}\{(s,t,e,\mathsf{hput}\ \alpha\ n;c)\}$	$\mathcal{C}\{(t(\alpha), q' \mid q{:}\mathsf{h}(n))\}$	$\mathcal{P}\{(s,t,e,c)\}$
$\mathcal{C}\{(t(\beta), q' \mid q)\}$	$\mathcal{P}\{(s,t,e,\mathsf{hcase}\ \beta\ \{c_i\})\}$	$\mathcal{C}\{(t(\beta), q' \mid q{:}(s,t,e,\mathsf{hc}\{c_i\}))\}$	$\mathcal{P}\{\}$

Table 9.6: Process execution steps (α with input polarity)

$\mathcal{C}\{(\beta, q{:}v \mid \mathsf{g}(s,t,e,c))\}$	$\mathcal{P}$	$\mathcal{C}\{(\beta, q \mid \varepsilon)\}$	$\mathcal{P}\{(v{:}s,t,e,c)\}$
$\mathcal{C}\{(t(\beta), q{:}\mathsf{h}(i) \mid (s,t,e,\mathsf{hc}\ \{c_j\}))\}$	$\mathcal{P}$	$\mathcal{C}\{(\beta, q \mid \varepsilon)\}$	$\mathcal{P}\{(s,t,e,c_i)\}$
$\mathcal{C}\{(\beta, \langle\beta_1,\beta_2\rangle \mid \begin{bmatrix}t,e,\alpha_1/\Gamma_1.c_1\\\alpha_2/\Gamma_2.c_2\end{bmatrix}\}$	$\mathcal{P}$	$\mathcal{C}\{\}$	$\mathcal{P}\left\{\begin{matrix}([],t_{\Gamma_1}[\beta_1/\alpha_1],e,c_1),\\([],t_{\Gamma_2}[\beta_2/\alpha_2],e,c_2)\end{matrix}\right\}$
$\mathcal{C}\{(\beta, \mathsf{close} \mid \mathsf{halt})\}$	$\mathcal{P}$	$\mathcal{C}\{\}$	$\mathcal{P}$

Table 9.7: Channel Manager's Actions for Non-Service Channels

9.4.6 Channel Manager's Actions

Table 9.7 describes how CAMPL deals with the dual command pairs of get-put, hput-hcase, split-fork and close-halt when get occurs before put, hcase occurs before hput, fork occurs before split. Note that Table 9.7 shows only half of the possible configurations of the Channel Manager for the above mentioned commands as the table is symmetric when input and output queues are swapped.

These commands are discussed with respect to the non-service channels. Channel Manager's actions for service channels are discussed in Section 9.4.6.

An explanation of the Channel Manager's actions is provided below:

- **Communication of values or a handle**: When a value/handle is waiting on a channel and there is a suspended process waiting for the value then one can transmit the value and re-enable the process which was waiting. Row 1 of Table 9.7 shows the communication of values. Row 2 shows the communication of a handle facilitated by the Channel Manager.

- **Split/Fork communication**: When a suspended process which wishes to fork is waiting on a channel and the corresponding split is made on the channel one assigns new global channel names to the channels introduced by the fork and enable the processes which are forked modifying the translations of the processes. Row 3 of Table 9.7 shows the split − fork communication facilitated by the Channel Manager.

- **Close/Halt communication**: A channel can be completely removed from the channel manger only when a close command matches a halt command. This is shown by the row 4 of Table 9.7.

Channel Manager's Actions for Service Channels

Service channels are built in channels in MPL that allow an MPL program to interact with the outside world. Service channels are slightly different from non-services channels in that a non-service channel connects MPL's processes while a service channel connects an MPL process to a built-in process with communication to outside world.

The standard way to use a service channel is to hput a handle on it and then do any of the get, put, close, or halt actions. A protocol commonly used with an output service channel is defined below:

```
protocol IntTerm (A) => P =
    GetInt   :: Get (A|P) => P
    PutInt   :: Put (A|P) => P
    Close    :: Top       => P
```

$\mathcal{C}\{(\beta, \mathsf{g}(s,t,e,c) : \varepsilon \mid \varepsilon : \mathsf{h}(1))\}$	$\mathcal{P}$	$\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon)\}$	$\mathcal{P}\{(v{:}s,t,e,c)\}$
$\mathcal{C}\{(\beta, \mathsf{v} : \varepsilon \mid \varepsilon : \mathsf{h}(2))\}$	$\mathcal{P}$	$\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon)\}$	$\mathcal{P}$
$\mathcal{C}\{(\beta, \mathsf{close} : \varepsilon \mid \varepsilon : \mathsf{h}(3))\}$	$\mathcal{P}$	$\mathcal{C} \setminus \beta$	$\mathcal{P}$
$\mathcal{C}\{(\beta, \mathsf{halt} : \varepsilon \mid \varepsilon : \mathsf{h}(3))\}$	$\mathcal{P}$	$\mathcal{C} \setminus \beta$	$\mathcal{P}$

Table 9.8: Channel Manager's Actions for Service Channels(Output Channel β)

where the name of the protocol is `IntTerm` and one can recursively get, put, and close a service channel using the handles `GetInt`, `PutInt`, and `Close` respectively.

The Channel Manager's actions for an output service channel are described in Table 9.8. In Table 9.8, $\mathsf{h}(1)$, $\mathsf{h}(2)$ and $\mathsf{h}(3)$ refer to the first, second and the third handle respectively.

Example: Executing Concurrent MPL Programs

This section shows an example of compiling and executing Concurrent MPL commands on CAMPL. In Table 9.9 a simple MPL program is compiled to CAMPL code. The MPL program consists of just one process, the main process. The main process takes two numbers on an output service channel named `intTerm1`, sums the numbers, and displays them on the same service channel.

In Table 9.10 and 9.11, the CAMPL code from Table 9.9 is executed stepwise based on the actions from Table 9.8 and 9.6, till the final state is reached. If the program had non-service channels in addition to the service channels, then Table 9.7 detailing the Channel Manager's actions for non-service channels would be used as well.

In Table 9.10 and 9.11, the *Translation* is represented by Γ which is a mapping of local channels to global channels. The translation $\Gamma = \big[\big(\ 0,\ 0\big), \big(\text{-}1,\ \text{-}1\big)\big]$ means that channels known locally as 0 and `-1` are known globally by the same numbers. This is because all the channels in the given program are service channels. Service channels are global channels and are known in every process by the same numbers. They are represented by numbers less than or equal to zero. zero is used for a special service channel known as `console`.

The first row of Table 9.10 describes the initial state in the execution of the main process.

Before	After

`protocol IntTerm (A) => P =`
`  GetInt :: Get (A|P) => P`
`  PutInt :: Put (A|P) => P`
`  Close :: Top => P`
 	
`coprotocol CP => Console (A) =`
`  GetIntC :: CP => Put (A|CP)`
`  PutIntC :: CP => Get (A|CP)`
`  CloseC :: CP => TopBot`
 	
`run console => intTerm1 -> do`
`  hput GetInt on intTerm1`
`  get num1 on intTerm1`
`  hput GetInt on intTerm1`
`  get num2 on intTerm1`
`  hput PutInt on intTerm1`
`  put (num1+num2) on intTerm1`
`  hput Close on intTerm1`
`  close intTerm1`
`  hput CloseC on console`
`  halt console`
	-- put first handle on channel -1 which is a service
	-- channel called intTerm1.First handle suggests
	-- get action on -1
	`hput -1 1`
	`get -1` *-- get a value on channel -1*
	`store` *-- store the value obtained from -1*
	-- get another value from -1 and store
	`hput -1 1`
	`get -1`
	`store`
	`access 2` *-- load the second number*
	`access 1` *-- load the first number*
	`add` *-- add the two numbers*
	
	-- put second handle on channel -1. Second Handle
	-- suggests a value will be put on -2
	`hput -1 2`
	`put -1` *-- put the sum on channel -1*
	
	-- put third handle on channel -1. Handle 3
	-- suggests that channel will be closed.
	`hput -1 3`
	`close -1` *-- close channel -1*
	`hput 0 3` *-- put third handle on channel 0*
	`halt 0` *-- halt channel 0*

Table 9.9: Example : Compilation of Concurrent MPL Program to CAMPL

Notice the values of different columns in the first row. The Channel Manager starts with empty queues for all the channels in the main process. *Stack* and *Environment* are empty and *Translation* is loaded with local to global mapping of channels of the main process. *Code* is initialised to the compiled CAMPL code for the process generated in Table 9.9.

Last row of Table 9.11 describes the final state in the execution of the main process. Every column is empty except *Stack* which contains the sum of the numbers (output) as its top element.

Channel Manager	Stack	Trans	Env	Code
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon)\end{array}\right\}$	ε	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	ε	hput -1 1 : c_1
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon : \mathrm{h}(1))\end{array}\right\}$	ε	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	ε	get -1 : c_2
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \mathsf{g}(\varepsilon, \Gamma, \varepsilon, c_2) : \varepsilon \mid \varepsilon : \mathsf{h}(1))\end{array}\right\}$	ε	ε	ε	ε
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon)\end{array}\right\}$	$v_1 : \varepsilon$	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	ε	store : c_3
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon)\end{array}\right\}$	ε	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	$v_1 : \varepsilon$	hput -1 1 : c_4
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon : \mathrm{h}(1))\end{array}\right\}$	ε	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	$v_1 : \varepsilon$	get -1 : c_5
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \mathsf{g}(\varepsilon, \Gamma, v_1 : \varepsilon, c_5) : \varepsilon \mid \varepsilon : \mathsf{h}(1))\end{array}\right\}$	ε	ε	ε	ε
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon)\end{array}\right\}$	$v_2 : \varepsilon$	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	$v_1 : \varepsilon$	store : c_6
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon)\end{array}\right\}$	ε	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	$v_2 : v_1 : \varepsilon$	access 2 : c_7
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon)\end{array}\right\}$	$v_1 : \varepsilon$	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	$v_2 : v_1 : \varepsilon$	access 1 : c_8
$\mathcal{C}\left\{\begin{array}{l}(0, \varepsilon \mid \varepsilon),\\ (\text{-}1, \varepsilon \mid \varepsilon)\end{array}\right\}$	$v_2 : v_1 : \varepsilon$	$\Gamma\left[\begin{array}{l}(0,\ 0),\\ (\text{-}1,\ \text{-}1)\end{array}\right]$	$v_2 : v_1 : \varepsilon$	add : c_9

Table 9.10: Executing Code on CAMPL (Continued On Table 9.11)

Channel Manager	Stack	Trans	Env	Code
$\mathcal{C}\left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (\text{-}1, \varepsilon \mid \varepsilon) \end{array} \right\}$	v_1+$v_2 : \varepsilon$	$\Gamma \left[\begin{array}{l} (0,\ 0), \\ (\text{-}1,\ \text{-}1) \end{array} \right]$	$v_2 : v_1 : \varepsilon$	hput -1 2 : c_{10}
$\mathcal{C}\left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (\text{-}1, \varepsilon \mid \varepsilon : \mathsf{h}(2)) \end{array} \right\}$	v_1+$v_2 : \varepsilon$	$\Gamma \left[\begin{array}{l} (0,\ 0), \\ (\text{-}1,\ \text{-}1) \end{array} \right]$	$v_2 : v_1 : \varepsilon$	put -1 : c_{10}
$\mathcal{C}\left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (\text{-}1, v_1\text{+}v_2 : \varepsilon \mid \varepsilon : \mathsf{h}(2)) \end{array} \right\}$	ε	$\Gamma \left[\begin{array}{l} (0,\ 0), \\ (\text{-}1,\ \text{-}1) \end{array} \right]$	$v_2 : v_1 : \varepsilon$	c_{10}
$\mathcal{C}\left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (\text{-}1, \varepsilon \mid \varepsilon) \end{array} \right\}$	ε	$\Gamma \left[\begin{array}{l} (0,\ 0), \\ (\text{-}1,\ \text{-}1) \end{array} \right]$	$v_2 : v_1 : \varepsilon$	hput -1 3 : c_{11}
$\mathcal{C}\left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (\text{-}1, \varepsilon \mid \varepsilon : \mathsf{h}(3)) \end{array} \right\}$	ε	$\Gamma \left[\begin{array}{l} (0,\ 0), \\ (\text{-}1,\ \text{-}1) \end{array} \right]$	$v_2 : v_1 : \varepsilon$	close -1 : c_{12}
$\mathcal{C}\left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (\text{-}1, \mathsf{close} : \varepsilon \mid \varepsilon : \mathsf{h}(3)) \end{array} \right\}$	ε	$\Gamma \left[\begin{array}{l} (0,\ 0), \\ (\text{-}1,\ \text{-}1) \end{array} \right]$	$v_2 : v_1 : \varepsilon$	c_{12}
$\mathcal{C}\left\{ (0, \varepsilon \mid \varepsilon) \right\}$	ε	$\Gamma \left[(0,\ 0) \right]$	$v_2 : v_1 : \varepsilon$	hput 0 3 : c_{13}
$\mathcal{C}\left\{ (0, \varepsilon \mid \varepsilon : \mathsf{h}(3)) \right\}$	ε	$\Gamma \left[(0,\ 0) \right]$	$v_2 : v_1 : \varepsilon$	halt 0 : ε
$\mathcal{C}\left\{ (0, \mathsf{halt} : \varepsilon \mid \varepsilon : \mathsf{h}(3)) \right\}$	ε	$\Gamma \left[(0,\ 0) \right]$	$v_2 : v_1 : \varepsilon$	ε
$\mathcal{C}\{\ \}$	ε	$\Gamma[\ ]$	ε	ε

Table 9.11: Executing Code on CAMPL (Continued from Table 9.10)

Bibliography

[1] Cockett, R., Pastro, C. (2009), *The logic of message passing*, Science of Computer Programming, Volume 74 Issue 8, pp. 498-533, Elsevier.

[2] Harper, R. (2000), *Programming in Standard ML*, Retrieved from https://www.cs.cmu.edu/ rwh/introsml

[3] Meijer, E., and Jeuring, J. (9915). *Merging Monads and Folds for Functional Programming*, Lecture Notes in Computer Science (LNCS), Volume 925, Springer.

[4] Cockett, R., and Fukushima T. (1992). *About Charity*, Yellow Series Report, No. 92/480/18.

[5] Cockett, R. (2016), *The Typed Lambda Calculus with Fixed Points*, Course Notes for Foundations of Functional Programming, pp. 12-18, Retrieved from http://pages.cpsc.ucalgary.ca/ robin/class/521/types/fixtypes.pdf

[6] Johnsson, T. (1985). *Lambda Lifting: Transforming Programs to Recursive Equations*,Proc. of a conference on Functional programming languages and computer architecture, pp. 190-203, Springer-Verlag.

[7] Landin, P.J. (1964). *The Mechanical Evaluation of Expressions*, The Computer Journal, Volume 6, Issue 4, pp. 308320

[8] Leroy, X. (2008). *Functional programming languages: Part II: abstract machines* , MPRI course 2-4-2, Functional programming languages, Retrieved from http://gallium.inria.fr/ xleroy/mpri/progfunc/

[9] Cardelli, L. (1983). *The Functional Abstract Machine*, Technical Report TR-107, Bell Labs.

[10] Yasin, M. (2012), *Linear Functors and their Fixed Points*, Retrieved from https://pages.cpsc.ucalgary.ca/ robin/Theses/theses.html

[11] Chakraborty, S. (2014), *Linear Functors and their Fixed Points*, Retrieved from https://pages.cpsc.ucalgary.ca/ robin/Theses/theses.html

[12] Cousineau, G., Curien, P.-L., and Mauny, M. (1987). *The categorical abstract machine*, Science of Computer Programming, Volume 8, Issue 2, pp. 173-202, Elsevier.

[13] Fairbairn, J., and Wray, S. (1987). *TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators*, Proc. Of a Conference on Functional Programming Languages and Computer Architecture, pp. 34-45, Springer-Verlag.

[14] Augustsson, L. (1985). *Compiling Pattern Matching*, Proc. of a conference on Functional Programming Languages and Computer Architecture, pp. 368-381, Springer-Verlag.

[15] Barrett, G., Wadler,P. (1986), *Derivation of a Pattern-Matching Compiler*, Retrieved from http://homepages.inf.ed.ac.uk/wadler/topics/language-design.html

[16] Jones, S.P. (1983). *The Functional Abstract Machine*, Journal of Functional Programming, Volume 2, pp. 127-202, Cambridge University Press.

[17] Tim, L., Yellin,F. (1999). *Java Virtual Machine Specification*, Second Edition, Addison-Wesley Longman Publishing Co.

[18] Aït-Kaci, H. (1991). *Warren's Abstract Machine: A Tutorial Reconstruction*, The MIT Press.

[19] Milner, R., Parrow, J., and Walker, D. (1992). *A calculus of mobile processes, part I/II*, Information and Computation, Volume 100, Issue 1, pp. 1-77, Academic Press.

[20] Milner, R. (1999). *Communicating and Mobile Systems: the pi-calculus*, Cambridge University Press.

[21] Vasconcelos, V.T. (2012). *Fundamentals of session types*, Information and Computation, Volume 217, pp. 52-70, Academic Press.

[22] Honda, K., Vasconcelos, V.T., and Kubo, M. (1998). *Language primitives and type disciplines for structured communication-based programming*, European Symposym on Programming, LNCS, Volume 1381, pp. 122-138, Springer-Verlag.

[23] Takeuchi, K., Honda, K., and Kubo, M. (1994). *An interaction-based language and its typing system*, European Symposym on Programming, LNCS, Volume 817, pp. 398-413, Springer-Verlag.

[24] Gay, S.J., Hole, M.J. (2005). *Subtyping for session types in the pi calculus*,Acta Informatica, Volume 42, Issue 2, pp. 191-225, Springer-Verlag.

[25] Honda, K., Yoshida, N., and Carbone, M. (2008). *Multiparty asynchronous session types*,POPL '08, Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 273-284.

[26] Baltazar, P., and Mostrous, D., Vasconcelos, V.T. (2012). *Linearly refined session types*, Electronic Proceedings in Theoretical Computer Science, 101, pp 38-49.

[27] Freeman, T., and Pfenning, F. (1991). *Refinement types for ML*, Programming Language Design and Implementation, Volume 26, pp 268-277, ACM.

[28] Honda, K., Mukhamedov, A., Brown, G., Chen, Tzu-Chun, and Yoshida, N. (2011). *Scribbling interactions with a Formal Foundation*, Distributed Computing and Internet Technology, LNCS, pp 55-75, ACM.

[29] Yoshida, N., Hu, R., Neykova, R., and Ng, N. (2014). *The Scribble Protocol Language*, LNCS, Volume 8358, pp 22-41, Springer.

[30] Neubauer, M., and Thiemann, P. (2004). *An Implementation of Session Types*, LNCS, Volume 3057, pp 56-70, Springer.

[31] Sackman, M., and Eisenbach, S. (2004). *Session Types in Haskell: Updating message passing for the 21st century*, Technical report, Imperial College, Department of Computing.

[32] Bhargavan, K., Corin, R., Denielou,Pierre-Malo, Fournet, C., and Leifer, J.J. (2009). *Cryptographic protocol synthesis and verification for multipary sessions*, Computer Security Foundations Symposium, pp 124-140, IEEE.

[33] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S. (2006). *Language Support for Fast and Reliable Message-based Communication in Singularity OS*, Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06, pp 177-190, ACM.

[34] Hunt, G., Larus, J.R., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B.D. (2005). *An overview of the singularity project*, Technical Report, Microsoft Research, 2005.

[35] Hu, R., Yoshida, N., Honda, K. (2008). *Session-based distributed programming in Java* , ECOOP '08 Proceedings of the 22nd European conference on Object-Oriented Programming, pp 516-541, Springer-Verlag.

[36] Ng, N., Yoshida, N., Honda, K. (2012). *Multiparty Session C: Safe Parallel Programming with Message Optimisation* , LNCS, Volume 7304, pp 202-218, Springer-Verlag.

[37] Pierce, B.C., Turner, D.N. (2000). *Pict: a programming language based on the Pi-Calculus* , Proof, language, and interaction, Volume 7304, pp 455-494, MIT Press.

[38] Vasconcelos, V.T. (1994). *Typed concurrent objects* , LNCS, Volume 821, pp 100-117, Springer.

[39] Hewitt, C., Bishop, P., Steiger, R. (1973). *A universal modular ACTOR formalism for artificial intelligence* , IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence, pp 235-245, Morgan Kaufmann Publishers Inc.

[40] Vasconcelos, C. (2016). *A Revision of the Mool Language* , arXiv:1604.06245.

Appendix A

MPL Programs

This appendix consists of three MPL programs.

A.1 Ticket Server Application

In this MPL program, there is a ticket server with two booking clients attached to it via channels. The ticket server has a certain number of tickets, input by the user, which the clients can book. To begin with, the first client books some tickets. The number of tickets booked by the first client is deducted from the total number of tickets that the server has. The server updates the number of tickets which it then passes to the second client. The second client books some tickets and updates the number of tickets that the server has. Thus, the ticket server alternates between the two clients till the number of tickets it holds becomes zero. The program performs some error checks as well: it doesn't allow any of the client to book more ticket than available and if any client tries to do so, the program discards that entry and prompts the client to re-enter the number of tickets.

```
-- ------------------------------------------------------------------------
-- ----------------------TICKET SERVER-------------------------------------
```

```
proc ticketServer :: Int,Int,Int,Int|
                     Console(Int),Console(Int) => IntTerm(Int) =
  tot,i,tot1,tot2|ch1,ch2 => i3
    do hput PutInt on i3
       put tot on i3     -- show the status of the server
       case i%2 == 1 of
         True
           do hput PutIntC on ch1
              put tot on ch1      -- pass the number of tickets to the client1
              hput GetIntC on ch1
              get tick1 on ch1    -- client1 books tickets
              case (tot-tick1) <= 0 of
                False
                  do ticketServer(tot-tick1,i+1,tot1+tick1,tot2|ch1,ch2 =>i3)
                True
                  do case (tot-tick1) == 0 of
                       True
                         do hput PutInt on i3
                            put (tot1+tick1) on i3
                            hput PutInt on i3
                            put tot2 on i3
                            hput PutIntC on ch1
                            put tot1 on ch1
                            hput Close on i3
                            close i3
                            hput CloseC on ch1
```

```
                    close ch1
                    hput CloseC on ch2
                    halt ch2
               False
                 do ticketServer(tot,i,tot1,tot2|ch1,ch2 => i3)
False
  do hput PutIntC on ch2 --pass the number of tickets to the client2
     put tot on ch2
     hput GetIntC on ch2
     get tick2 on ch2   -- client2 books tickets
     case (tot-tick2) <= 0 of
       False
         do ticketServer(tot-tick2,i+1,tot1,tot2+tick2|ch1,ch2 => i3)
       True
         do case (tot-tick2) == 0 of
              True
                do hput PutInt on i3
                   put tot1 on i3
                   hput PutInt on i3
                   put(tot2 + tick2) on i3
                   hput Close on i3
                   close i3
                   hput CloseC on ch1
                   close ch1
                   hput CloseC on ch2
                   halt ch2
              False
```

```
                        do ticketServer(tot,i,tot1,tot2|ch1,ch2 => i3)

proc server_Main :: Console(Int) (*) Console(Int) => IntTerm(Int) =
  | ch => i3
     do split ch into ch1,ch2
        hput GetInt on i3
        get tot on i3
        ticketServer(tot,1,0,0| ch1,ch2 => i3)

-- ------------------------------------------------------------------
-- -------------------------TICKET CLIENTS-----------------------------

proc ticketClients ::  |  => IntTerm(Int),Console(Int) =
  | => i1,ch
     do hcase ch of
          GetIntC
            do hput GetInt on i1   -- get number of tickets to be booked
               get bookTicks on i1
               put bookTicks on ch
               ticketClients(|=> i1,ch)
          PutIntC
            do get remTicks on ch -- get remaining tickets from the server
               hput PutInt on i1  -- display the number of available tickets
               put remTicks on i1
               ticketClients(| => i1,ch)
          CloseC
            do close ch
```

```
                hput Close on i1
                halt i1


proc clientFrontEnd :: | Console(Int) => IntTerm(Int), IntTerm(Int),
                                         Console(Int) (*) Console(Int) =
  | console => i1,i2,ch
    do hput CloseC on console
       close console
       fork ch as
         ch1
           do ticketClients(| => i1,ch1)
         ch2
           do ticketClients(| => i2,ch2)

-- ------------------------------------------------------------------------
-- -----------------------------MAIN PROCESS-----------------------------


run :: Console(Int) => IntTerm(Int), IntTerm(Int), IntTerm(Int) =
  console => intTerm1,intTerm2,intTerm3
    do plug
         clientFrontEnd(|console => intTerm1,intTerm2,ch)
         server_Main(|ch => intTerm3)
```

A.2 Playing Tic-Tac-Toe

This program implements the game of tic-tac-toe between two players. The two players enter their moves, between the numbers one and nine, on their respective terminals. At any point in the game, the players can see the status of their board. Once a player makes a move, the status of the board is updated. If a player wins as a result of the move, the winning message along with the status of the board is printed. If the match ends in a draw, then the relevant message is printed. The program does some error checking: one can't repeat a move that has already been played and one can't enter a number less than one or greater than nine.

```
-- ------------------------------------------------------------------------
-- ---------------(CO)DATA/(CO)PROTOCOL DEFINITIONS-----------------------
data Status -> C = Cont :: -> C
                   Draw :: -> C
                   Win  :: -> C

data Row -> C = Triple :: String,String,String -> C

data Board -> C = Piece :: [Row] -> C

protocol ProtMoves (A) => P =
    GetMove :: Get (A|P) => P
    CloseM  :: TopBot    => P

-- ------------------------------------------------------------------------
-- --------------------GENERAL HELPER FUNCTIONS---------------------------

-- This function checks if a given element is present in the list
```

```
elem :: Int,[Int] -> Bool =
  v,[]   -> False
  v,x:xs -> case x == v of
              True  -> True
              False -> elem(v,xs)

-- ------------------------------------------------------------------------
-- ---------------FUNCTIONS USED IN GAME LOGIC----------------------------

--check if a move is a valid and a new move
isRightMove :: Int,[Int] -> Either(String,Bool) =
  num,list ->  case num <= 9 of
                 True -> case num <= 0 of
                           True  -> Left("Move less than 0")
                           False -> case elem(num,list) of
                                      True  -> Left("Move repeated")
                                      False -> Right(True)
                 False -> Left("Move greater than 9")



-- checks if the first list is a part of the second list
isPartOf :: [Int],[Int] -> Bool =
  [],sList -> True
  (f:fs),sList -> case elem(f,sList) of
                    True  -> isPartOf(fs,sList)
                    False -> False
```

```
isSol :: [Int],[[Int]] -> Bool =
  moves,[] -> False
  moves,(s:ss) -> case isPartOf(s,moves) of
                    True  -> True
                    False -> isSol(moves,ss)


getStatus :: [Int],Int -> Status =
  moves,count -> case count == 9 of
                   True ->  case vict of
                              True  -> Win
                              False -> Draw
                   False -> case vict of
                              True  -> Win
                              False -> Cont
             where
               vict    = isSol(moves,solList)
               solList = [[1,5,9],[3,5,7],[1,2,3],[4,5,6],
                          [7,8,9],[1,4,7],[2,5,8],[3,6,9]]


errorMsg :: Int,String -> [Char] =
  num,emsg -> case num == 1 of
                True  -> unstring(xmsg)
                False -> unstring (ymsg)
```

```
            where
              xplay = "(by player X) /Try Again.\n"
              yplay = "(by player O) /Try Again.\n"
              xmsg  = concat(emsg,xplay)
              ymsg  = concat(emsg,yplay)


winDrawMsg :: Bool,String -> [Char] =
  win,player -> case win of
                  True -> case eqS(player,"X") of
                            True  -> helperFun("X WINS")
                            False -> helperFun("O WINS")
                  False -> helperFun("MATCH DRAWN")
            where
              helperFun :: String -> [Char] =
                msg -> unstring (concatList([msg,"\n",stars]))


turnMsg :: String -> [Char] =
  player -> switch
              eqS(player,"X") = unstring(concatList([stars,"X's Turn",stars]))
              default = unstring(concatList([stars,"O's Turn",stars]))


-- ------------------------------------------------------------------------
-- -------------------FUNCTIONS FOR BOARD REPRESENTATION-------------------
```

```
-- Representation of an empty Board
empBoard :: () -> Board  =
   -> Piece ([emptyRow,emptyRow,emptyRow])
    where
       emptyBox = "|   |"
       emptyRow = Triple(emptyBox,emptyBox,emptyBox)



-- Below function change the representation of a board based on the move made
replaceRow :: Row,Int,String -> Row =
  Triple (a,b,c),colN,newP ->
    case colN == 0 of
      True  -> Triple (newP,b,c)
      False -> case colN == 1 of
                 True  -> Triple(a,newP,c)
                 False -> Triple(a,b,newP)

-- modify the representation of board based on the input
newBdRep :: Board,Int,Int -> Board =
  Piece (rows),move,flag ->
      case flag == 0 of
        True  -> Piece(newB_Help(rows,quot,rem,0,box_O,[]))
        False -> Piece(newB_Help(rows,quot,rem,0,box_X,[]))
    where
      box_X = "| X |"
      box_O = "| O |"
      nMove = move -1
```

```
      quot  = quotI(nMove,3)
      rem   = nMove % 3

      newB_Help :: [Row],Int,Int,Int,String,[Row] -> [Row] =
        rList,r,c,count,newP,addR ->
          case rList of
            []   -> []
            x:xs -> case r == count of
                      True  -> addR ++ ((replaceRow(x,c,newP)):xs)
                      False -> newB_Help(xs,r,c,count+1,newP,addR ++ [x])

finBdRep :: Board,Int,Int -> Board =
  board,oMv,xMv -> newBdRep(newBdRep(board,oMv,0),xMv,1)



hLine :: () -> String =
    -> " -------------\n"

stars :: () -> String =
  -> "\n********************\n"

addNL :: String -> String =
  piece  -> concat(piece,"\n")

addHL :: String -> String =
  piece  -> concat(hLine,piece)
```

```
repRow :: Row -> String =
  Triple (s1,s2,s3) -> concat(concat(ns1,s2),addNL(s3))
    where
      ns1 = addHL(s1)

drawBoardHelp :: [Row] ,String -> String =
  [],list -> concatList ([list,hLine,stars])
  (row:rest),list -> drawBoardHelp (rest,newList)
     where
       rowStr = repRow(row)
       newList = concat(list,rowStr)

drawBoard :: Board -> [Char] =
  Piece(rList) -> unstring (drawBoardHelp (rList,""))

-- Modify the board based on the move
drawMdfdBd :: Board,Int,Int -> [Char] =
  board,move,flag -> drawBoard(newBdRep(board,move,flag))

drawFnBd :: Board,Int,Int -> [Char] =
   board,mvO,mvX -> drawBoard(finBdRep(board,mvO,mvX))

-- ------------------------------------------------------------------
-- -------------------------PROCESSES--------------------------------

proc printString :: [Char] |  => IntTerm(Char) =
  charList |  => c1
```

```
   do case charList of
        []
          do hput Close on c1
             halt c1
        x:xs
          do hput PutInt on c1
             put x on c1
             printString(xs | => c1)

proc playerShow :: String, [Char] | IntTerm([Char]) => IntTerm(Char) =
  mode,cList | pch => c1
    do case eqS(mode,"receive") of
          True
            do hcase pch of
                 GetInt
                   do put [' '] on pch
                      playerShow(mode,cList|pch => c1)
                 PutInt
                   do get board on pch
                      playerShow("show",board|pch => c1)
                 Close
                   do hput Close on c1
                      close c1
                      halt pch
          False do
            do case cList of
                 []
```

```
                  do playerShow("receive",[' ']|pch => c1)
                x:xs
                  do hput PutInt on c1
                     put x on c1
                     playerShow("show",xs|pch  => c1)

-- ------------------------------------------------------------------
-- ----------------------------PLAYER 1-----------------------------

proc closeP1 :: Bool, String | Console(T0) =>
                IntTerm(Int),IntTerm([Char]), ProtMoves(Int) =
  win,player | cco => i1,pch,ch
   do hput PutInt on pch
      put winDrawMsg(win,player) on pch
      hput CloseM on ch
      close ch
      hput Close on pch
      close pch
      hput Close on i1
      close i1
      hput CloseC on cco
      halt cco


proc p1Get :: [Int], [Int], Board, Bool, Int |
              Console(Char) => IntTerm(Int),IntTerm([Char]),ProtMoves(Int) =
   xs,os,bd,xRep,ct| cco => i1,pch,ch
```

```
do case xRep of
     True
       do hput PutInt on pch
          put turnMsg("X") on pch
          hput GetInt on i1
          get xMv on i1
          case isRightMove(xMv,append(xs,os)) of
            Left(emsg)
              do hput PutInt on pch
                 put errorMsg(1,emsg) on pch
                 p1Get (xs,os,bd,True,ct|cco=> i1,pch,ch)
            Right(bool1)
              do hput PutInt on pch
                 put drawMdfdBd (bd,xMv,1) on pch
                 case getStatus (xMv:xs,ct+1) of
                   Draw
                     do closeP1(False,"X"|cco => i1,pch,ch)
                   Win
                     do closeP1(True ,"X"|cco => i1,pch,ch)
                   Cont
                     do p1Get(xMv:xs,os,newBdRep(bd,xMv,1),False,ct+1|
                              cco=> i1,pch,ch)
     False
       do hput PutInt on pch
          put turnMsg("O") on pch
          hput GetMove on ch
          get oMv on ch
```

```
case isRightMove (oMv,append(xs,os)) of
  Left(emsg)
    do hput PutInt on pch
       put errorMsg(2,emsg) on pch
       p1Get(xs,os,bd,xRep,ct| cco => i1,pch,ch)
  Right(bool)
    do hput PutInt on pch
       put drawMdfdBd (bd,oMv,0) on pch
       case getStatus (oMv:os,ct+1) of
         Draw
           do closeP1(False,"O" |cco => i1,pch,ch)
         Win
           do closeP1(True,"O"  |cco => i1,pch,ch)
         Cont
           do hput PutInt on pch
              put turnMsg("X") on pch
              hput GetInt on i1
              get xMv on i1
              case isRightMove (xMv,append(xs,oMv:os)) of
                Left(emsg)
                  do hput PutInt on pch
                     put errorMsg(1,emsg) on pch
                     p1Get(xs,oMv:os,newBdRep(bd,oMv,0),
                           True,ct+1|cco=> i1,pch,ch)
                Right(bool)
                  do hput PutInt on pch
                     put drawFnBd (bd,oMv,xMv) on pch
```

```
                                    case getStatus (xMv:xs,ct+2) of
                                      Draw
                                        do closeP1(False,"X"  |cco => i1,pch,ch)
                                      Win
                                        do closeP1(True ,"X" |cco => i1,pch,ch)
                                      Cont
                                        do p1Get(xMv:xs,oMv:os,
                                                 finBdRep(bd,oMv,xMv),xRep,ct+2
                                                 |cco=> i1,pch,ch)


proc p1Init :: | Console(Char) =>
                 IntTerm(Int),IntTerm([Char]),ProtMoves(Int) =
  | cco => i1,pch,ch
  do hput GetInt on i1
     get xMv on i1
     case isRightMove (xMv,[]) of
       Left(emsg)
         do hput PutInt on pch
            put errorMsg(1,emsg) on pch
            p1Init(|cco=> i1,pch,ch)
       Right(bool)
         do hput PutInt on pch
            put drawMdfdBd (empBoard,xMv,1) on pch
            p1Get([xMv],[],newBdRep(empBoard,xMv,1),False,1|
                   cco=> i1,pch,ch)
```

```
proc player1 :: [Char] |
                Console(Char) => IntTerm(Int),IntTerm(Char),ProtMoves(Int)  =
  board |cco => i1,c1,ch
    do case board of
         []
           do plug
                p1Init(|cco => i1,pch,ch)
                playerShow("receive",[' '] | pch => c1)
         x:xs
           do hput PutInt on c1
              put x on c1
              player1(xs |cco => i1,c1,ch)

-- ------------------------------------------------------------------
-- -----------------------------PLAYER 2-----------------------------

proc player2 :: | ProtMoves(Int) => IntTerm(Int)=
  | ch => i2
    do hcase ch of
       -- player1 is asking player2 to make a move and put it on ch
         GetMove
           do hput GetInt on i2
              get move on i2
              put move on ch
              player2(|ch => i2)
         CloseM
```

```
            do hput Close on i2
               close i2
               halt ch

-- ------------------------------------------------------------------
-- ----------------------MAIN PROCESS--------------------------------

run :: Console(Char) => IntTerm(Char),IntTerm(Int),IntTerm(Int)=
  cconsole => charTerm1,intTerm1,intTerm2
    do plug
         player1(drawBoard(empBoard)|cconsole => intTerm1,charTerm1,ch)
         player2(| ch => intTerm2)
```

A.3 Chat Application

This program implements a chat application between two users. The program asks for user names to be entered on a terminal. These two users can then communicate on using their terminals. The first user starts the process of chatting and enters the message he wants to transmit followed by the return key. Once the return key is pressed, the message is transmitted and printed on the terminal of the second user. The second user can then type the message on its terminal and press the return key to transmit it. The users alternate between sending and receiving messages.

```
-- ------------------------------------------------------------------
-- ---------------(CO)DATA/(CO)PROTOCOL DEFINITIONS-----------------------

data Mode -> C =
```

```
    Snd  :: [Char] -> C
    Rcv  ::        -> C
    Prnt :: [Char] -> C

protocol OnlyGet (A) => P =
    JustGet :: Get (A|P) => P
    CloseJG :: TopBot    => P

protocol JustPut (A) => P =
    OnlyPut :: Get (A|P) => P

-- ------------------------------------------------------------------------
-- ----------------------FUNCTIONS-----------------------------------------

printName :: [Char] -> [Char] =
   name -> append(name,unstring(" says:"))

empStr :: () -> [Char] =
    -> [' ']

formatMsg :: [Char],[Char] -> [Char] =
    msg,name -> append(printName(name),reverse(msg))

salut :: [Char] -> [Char] =
    name -> append(inchs3,inchs2)
      where
        space  = "            "
```

```
        stars  = "*****************************************\n"
        inStr1 = concatList([space,stars,space,space])
        inStr2 = concatList(["\n",space,stars,"\n"])
        inchs1 = unstring (inStr1)
        inchs2 = unstring (inStr2)
        inchs3 = append(inchs1,name)

-- ------------------------------------------------------------------
-- ----------------------FIRST USER----------------------------------

proc chatter1 :: Mode, [Char] | Console(Char) =>
                                IntTerm(Char),IntTerm([Char]) =
  mode,name |cco => c1,ch
    do case mode of
         Snd(msg)
           do hput GetInt on c1
              get char on c1
              case eqC(char,'\n') of
                True
                  do hput PutInt on ch
                     put formatMsg(char:msg,name) on ch
                     chatter1(Rcv,name|cco => c1,ch)
                False
                  do chatter1(Snd(char:msg),name | cco => c1,ch)
         Rcv
           do hput GetInt on ch
              get prnMsg on ch
```

```
            chatter1(Prnt(prnMsg),name | cco => c1,ch)
        Prnt(msg)
          do case msg of
               []
                 do hput PutInt on c1
                    put '>' on c1
                    hput PutInt on c1
                    put ' ' on c1
                    chatter1(Snd([' ']),name|cco => c1,ch)
               x:xs
                 do hput PutInt on c1
                    put x on c1
                    chatter1(Prnt(xs),name|cco => c1,ch)

-- ------------------------------------------------------------------
-- ----------------------SECOND USER---------------------------------

proc chatter2 :: Mode, [Char], Int | IntTerm([Char]) => IntTerm(Char) =
 mode,name,flag | ch => c2
   do case mode of
        Snd(msg)
          do hput GetInt on c2
             get char on c2
             case eqC(char,'\n') of
               True
                 do hcase ch of
                      GetInt
```

```
                do put formatMsg(char:msg,name) on ch
                   chatter2(Rcv,name,flag|ch => c2)
             PutInt
                do get pmsg on ch
                   chatter2(Prnt(pmsg),name,flag| ch => c2)
             Close
                do hput Close on c2
                   close c2
                   halt ch
       False
         do chatter2(Snd(char:msg),name,flag|ch => c2)
Rcv
  do hcase ch of
       GetInt
         do put [' '] on ch
            chatter2(Snd([' ']),name,flag|ch => c2)
       PutInt
         do get prnMsg on ch
            chatter2(Prnt(prnMsg),name,flag|ch => c2)
       Close
         do hput Close on c2
            close c2
            halt ch
Prnt(msg)
  do case msg of
       []
         do hput PutInt on c2
```

```
                    put '>' on c2
                    hput PutInt on c2
                    put ' ' on c2
                    case flag == 0 of
                      True
                        do chatter2(Rcv,name,1 | ch => c2)
                      False
                        do chatter2(Snd([' ']),name,flag|ch => c2)
               x:xs
                 do hput PutInt on c2
                    put x on c2
                    chatter2(Prnt(xs),name,flag|ch => c2)


-- This process plugs chatter1 and chatter2 along a channel
proc startChat :: | Console(Char) =>
                    IntTerm(Char),IntTerm(Char),OnlyGet([Char]) =
  | cco => c1,c2,ch
    do hput JustGet on ch
       get usnm1 on ch
       hput JustGet on ch
       get usnm2 on ch
       hput CloseJG on ch
       close ch
       plug
         chatter1(Prnt(salut(usnm1)),usnm1|cco => c1,nch)
         chatter2(Prnt(salut(usnm2)),usnm2,0|nch =>c2)
```

```
-- This process get the user names for chatting
proc getUserName ::  [Char] | OnlyGet([Char]) => IntTerm(Char) =
  name | ch => c3
  do hput GetInt on c3
     get char on c3
     case eqC(char,'\n') of
       True
         do hcase ch of
              JustGet
                do put reverse(name) on ch
                   getUserName(empStr|ch => c3)
              CloseJG
                do hput Close on c3
                   close c3
                   halt ch
       False
         do getUserName(char:name|ch => c3)

-- ------------------------------------------------------------------
-- -----------------------------MAIN PROCESS-------------------------

run :: Console(Char) => IntTerm(Char),IntTerm(Char),IntTerm(Char) =
  cconsole => charTerm1,charTerm2,charTerm3
    do plug
         startChat(|cconsole => charTerm1,charTerm2,ch)
         getUserName(empStr|ch => charTerm3)
```

Appendix B

BNFC Grammar for MPL

B.1 MPL Program

An MPL program consists of a list of MPL statements followed by a *run* statement.

```
MPL -> [MPLStmt] RunStmt .
```

B.2 Run Statement

`RunStmt` is used to define the *main process* definition in an MPL program. The main process is the point of entry in an MPL program. It has syntax given by:

```
RunStmt  ->   "run" [Channel] "=>" [Channel] Process
           | "run" "::" [ChType] "=>" [ChType]
             [Channel] "=>" [Channel] Process.


Channel  -> Ident.
```

The main process is defined with the keyword `"run"`. `[Channel] "=>" [Channel]` is the list of input and output channel names, which are separated by `"=>"`. The main process can optionally be annotated with a process type as shown in the second rule for `RunStmt`.

`[ChType] "=>" [ChType]` is the process type of the main process. The elements on left side of `"=>"` represents the channel types for the input channels. Similarly, the ones on the right side represent the channel types for the output channels. A channel is an identifier starting with a lower case letter.

B.3 MPLStmt (An MPL Statement)

```
MPLstmt ->  "defn" [Defn] "where" [MPLstmt]
          | Defn .
```

An MPL statement can either be a list of definitions which require some local definitions as shown in the first rule or a bare definition, `Defn`, as shown in the second rule. `"defn"`, and `"where"` are layout keywords.

B.4 Defn (An MPL Definition)

An MPL definition, `Defn`, is either a data/codata definition, a protocol/coprotocol definition, a function definition, or a process definition.

```
Defn ->   SeqDataDefn
        | ConcDataDefn
        | FunctionDefn
        | ProcessDefn .
```

`SeqDataDefn`, which stands for a sequential data definition, is either a data type or a codata type definition. Similarly, a `ConcDataDefn` stands for a concurrent data type definition and is either a protocol or a coprotocol definition.

B.5 SeqDataDefn (Sequential Data Definition)

An MPL (co)data type is defined by the keyword `"data"` or `"codata"` followed by a list of (co)data clauses. These elements in the list of (co)data clauses are separated by the keyword `"and"` if mutually recursive (co)data types are being defined.

```
SeqDataDefn ->  "data"   [DataClause]
              | "codata" [CodataClause].
```

B.6 ConcDataDefn (Concurrent Data Definition)

An MPL (co)protocol is defined by the keyword `"protocol"` or `"coprotocol"` followed by a list of protocol clauses. These elements in the list of the protocol clauses are separated by the keyword `"and"` if mutually recursive (co)protocols are being defined.

```
ConcDataDefn ->   "protocol"   [ProtocolClause]
                | "coprotocol" [ProtocolClause].
```

B.7 FunctionDefn (Function Definition)

A sequential function definition has syntax given by:

```
FunctionDefn ->   "fun" Ident "::" [Type] "->" Type "=" [PattTermPharse]
                | "fun" Ident "=" [PattTermPharse].
```

`Ident` is the name of the function: an identifier starting with a lower case letter. A `FunctionDefn` can also be annotated with a function type: `[Type] "=>" Type`. `[PattTermPharse]` is a list of `PattTermPharse` arranged using layout syntax with respect to `"="`.

B.8 ProcessDefn (Process Definition)

A process definition has syntax given by:

```
ProcessDefn ->   "proc" Ident "::" [Type] "|" [ChanType] "=>"
                 [ChanType] "=" [PatProcessPhr]
               | "proc" Ident "=" [PatProcessPhr].
```

`Ident` is the name of the process: an identifier starting with a small letter. The `ProcessDefn` can also be annotated with a process type: `[Type] "|" [ChanType] "=>" [ChanType]`. `[Type]` represents the types of the sequential parameters, and `[ChanType] "=>" [ChanType]` represents the channel types of the channels of the process. `[PatProcessPhr]` is a list of patterns-process phrase arranged in a layout syntax with respect to the `"="`.

B.9 Data and Codata Clause

A data clause has syntax given by:

```
DataClause   -> DataProtName "->"  UIdent "=" [DataPhrase]
DataProtName -> UIdent "(" [UIdent] ")"
```

`DataProtName` is the name of the data type. `UIdent` is the name of the data definition, and `[UIdent]` is the list of type variables in which the data type is parametric. `UIdent` is a state variable: an identifier starting with a capital letter. `[DataPhrase]` is a list of constructor definitions. These constructors are arranged in a layout syntax with respect to `"="`. Similarly, a codata clause has syntax given by:

```
CodataClause -> UIdent "->" DataProtName "=" [CodataPhrase]
```

where `[CodataPhrase]` is a list of destructor definitions arranged in a layout syntax using `"="`.

B.10 ProtocolClause/CoprotocolClause

The protocol clause and the coprotocol clauses are defined as:

```
ProtocolClause   -> DataProtName "=>" UIdent "=" [ProtocolPhrase] .
CoprotocolClause -> UIdent "=>" DataProtName "=" [ProtocolPhrase] .
```

`DataProtName` is the name of the (co)protocol. `UIdent` is a state variable. It is an identifier starting with a upper case character. `[ProtocolPhrase]` is a list of (co)handles of the (co)protocol. These are arranged in a layout syntax with respect to `"="`.

B.11 DataPhrase/CodataPhrase

A data phrase in a data definition is the constructor name with its annotated type. It has syntax given by:

```
DataPhrase -> [UIdent] "::" [Type] "->"  UIdent.
```

`[UIdent]` is the list of constructor names. Constructor names start with a capital letter. Note that the above rule obviates the need of repeating the types in the data definition if there are multiple constructors with the same type. For example -

```
data Bool -> C = True,False :: -> C.
```

`[Type]` is the input type of a constructor. The state variable, `UIdent`, is on the right hand side of `->`. Similarly, a `CodataPhrase` in a codata defintion consists of destructor names annotated with their type. It has syntax given by:

```
CodataPhrase -> [Ident] "::" [Type] "->"  Type.
```

B.12 ProtocolPhrase/CoprotocolPhrase

A (co)protocol phrase defines the (co)handle of a protocol. It has syntax given by:

```
ProtocolPhrase -> UIdent "::" ChanType "=>" UIdent.
CoprotocolPhrase -> UIdent "::" UIdent   "=>" ChanType
```

First `UIdent` is the name of the (co)handle and the second `UIdent` is the state variable associated with the (co)handle. UIdent is an identifier starting with an upper case letter. `ChanType` is the channel type associated with the given (co)handle.

B.13 Type (Sequential Type)

A sequential type has syntax given by :

```
Type  ->   Type1 "=>" Type
         | Type1.

Type1 ->  "(" ")"
         | "[" Type1 "]"
         | UIdent "(" [Type] ")"
         | UIdent
         | "(" [Type] ")".
```

where:

- The type `Type1 "=>" Type` is the special syntax for the exponential codata type.
- `"(" ")"` is the unit type.
- `"[" Type1 "]"` is the special syntax for the list data type.

- The syntax of a (co)data type is given by `UIdent "(" [Type] ")"`. `UIdent` is the name of the (co)data type and `[Type]` is the list of the type arguments that the (co)data type takes.
- `UIdent` is an identifier starting with a capital letter: it can either represent a built in constant data type or a type variable.
- `"(" [Type] ")"` is a bracketed sequential type if the number of elements in the type list `[Type]` is one and a product type if the number of elements are greater than or equal to two.

B.14 ChanType (Concurrent Type)

A channel type (concurrent type) has syntax given by:

```
ChanType ->   ChanType1 "(*)" chanType
            | ChanType1 "(+)" ChanType.


Protocol1 ->   "Get" "(" Type "|" ChanType ")"
             | "Put" "(" Type "|" ChanType ")"
             | "Neg" "(" ChanType ")"
             | TopBot
             | UIdent "(" [Type] ")"
```

where:

- `ChanType1 "(*)" chanType` is the *Tensor* channel type.
- `ChanType1 "(+)" ChanType` is the *Par* channel type.
- `"Get" "(" Type "|" ChanType ")"` is the `Get` channel type.

- `"Put" "(" Type "|" ChanType ")"` is the `Put` channel type.
- `"Neg" "(" ChanType ")"` is the *Negation* of a channel type.
- `TopBot` is both the the *Top* and *Bottom* channel type.
- `UIdent "(" [Type] ")"` is a (co)protocol definitions. `UIdent` is the name of the (co)protocol and `[Type]` is the list of the type arguments that the (co)protocol definition takes.

B.15 PattTermPharse (Pattern-Term Phrase)

A `PattTermPharse` has syntax given by:

```
PattTermPharse ->  [Pattern] "->"  Term
                 | [Pattern] "->" "switch" [GuardedTerm] .
```

`Pattern` is a sequential pattern. `Term` is a sequential term. Terms can be defined with boolean guards with the `"switch"` keyword.

B.16 PatProcessPhr (Patterns-Channels-Process Commands Pharse)

A `PatProcessPhr` has syntax given by:

```
PatProcessPhr -> [Pattern] "|"  [Channel] "=>" [Channel] Process
```

`[Pattern]` is a list of sequential patterns. The two list of channel names separated by `"=>"` symbols are the input and the output channels. `Process` consists of a list of process commands.

B.17 Sequential Pattern

A sequential pattern has syntax given by:

```
Pattern ->   Pattern1 ":" Pattern
           | Pattern1.

Pattern1 ->  UIdent "(" [Pattern] ")"
           | "(" [DestPattPhr] ")"
           | UIdent
           | "[" [Pattern] "]"
           | Ident
           | "_"
           | "(" Pattern ")" .

DestPattPhr -> UIdent ":=" Pattern.
```

where:

- `Pattern1 ":" Pattern` is a constructor pattern for the list data type.
- `UIdent "(" [Pattern] ")"` is a constructor pattern which takes a list of patterns as arguments.
- `"(" [DestPattPhr] ")"` is a destructor pattern which consists of a list of destructor pattern phrases.
- `UIdent ":=" Pattern` is a destructor pattern where `UIdent` is the destructor name and `Pattern` is the corresponding patttern.

- The second rule of `Pattern1`, which is `UIdent`, represents a constructor/destructor pattern which does not take any arguments.
- `"[" [Pattern] "]"` is a pattern for the list type.
- The fifth rule of `Pattern1`, which is an identifier starting with a small letter, is the variable pattern.
- `"_"` is the don't care pattern.
- `"(" [Pattern] ")"` is a bracketed pattern if the number of elements in the pattern list `[Type]` is 1 and a product pattern if the number of elements are greater than or equal to 2.

B.18 Sequential Term

A sequential term has syntax given by:

```
Term ->   Term1 ":" Term
        | Term1 "where" [WhereDefn] .
        | Term1


Term1 ->   Term1 Infix0op Term2
         | Term2.

Term2 ->   Term2 Infix1op Term3
         | Term3.
```

```
Term3 ->   Term3 Infix2op Term4
         | Term4.

Term4 ->   Term4 Infix3op Term5
         | Term5

Term5 ->   Term5 Infix4op Term6
         | Term6

Term6 ->   Term6 Infix5op Term7
         | Term7.

Term7 ->   Term7 Infix6op Term8
         | Term8

Term8 ->   Term8 Infix7op Term9
         | Term9

Term9  ->   "[" [Term] "]"
          | Ident
          | ConstantType
          | "if" Term "then" Term "else" "" Term ""
          | "unfold" Ident "with" [FoldPattern]
          | "fold" Ident "of" [FoldPattern]
          | "fold" Term "of" [PattTermPharse]
          | UIdent "(" [Term] ")"
          | UIdent
```

```
          | Ident "(" [Term] ")"
          | "record" "of" [RecordEntry]
          | "(" [RecordEntry] ")"
          | "(" [Term] ")"

Infix0op -> "||".
Infix1op -> "&&".
Infix2op -> "=="| "/=" | '<' |'>'|"<="|">=".
Infix3op -> "++".
Infix4op -> '+'|'-'.
Infix5op -> '*'| '/' | '%' .
Infix6op -> '^'.
Infix7op -> "!!".

ConstantType ->   Integer
                | Double
                | Char
                | String
```

where:

- `Term1 ":" Term` is the special syntax for a `Cons` constructor of the list data type.
- `Term1 "where" "" [WhereDefns] ""` is the syntax for the `where` term. `WhereDefns` is a variable declaration, and the MPL declarations that can be made in the `where` part of the term.
- `Term`(n) `Infix`$(n-1)$`op Term`$(n+1)$ is the infix term corresponding to the n^{th} term.

- `"[" [Term] "]"` is a term of the list data type.
- `Ident` is a variable term.
- `ConstantType` is the constant term: integer, float, char, and strings.
- `UIdent "(" [Term] ")"` is the constructor terms with arguments.
- `UIdent` is the constructor term without arguments.
- `Ident "(" [Term] ")"` is function application: `Ident` a the name of the function being called and `[Term]` is a list of arguments with which the function is called.
- Record of a codata type can be created with one of the following syntax:
  - `"record" "of" [RecordEntry]` The list of `RecordEntry` is arranged in a layout syntax with respect to the `"of"` keyword.
  - `"(" [RecordEntry] ")"`
- `"(" [Term] ")"` is a bracketed term if the number of elements in the term list `[Term]` is one and a product term if the number of elements are greater than or equal to two.

B.19 GuardedTerm

A guarded term has syntax given by:

```
GuardedTerm  -> Term "=" Term
              | "default" "=" Term
```

where:

- `Term "=" Term` is the guarded term where the term on the left side of the `"="` symbol evaluates to a boolean value.
- `"default" "=" Term` is the default case when the left term of any other guarded term doesn't return `True`.

B.20 WhereDefn (Local Definitions in the Where Term)

The local definitions in the `where` terms are defined as:

```
WhereDefn ->  Defn
            | Ident "=" Term.
```

`Defn` is a local MPL definition. `Ident "=" Term` is a local variable decalaration. `Ident` is a variable name and `Term` is a sequential term assigned to the variable.

B.21 FoldPattern

A `FoldPattern` is one branch of the fold/unfold terms. It has syntax given by:

```
FoldPattern -> UIdent ":"  [Ident]  "="  Term.
```

`UIdent` is the name of a constructor/destructor of the data type for which the `fold/unfold` term is being written. `[Ident]` is the argument corresponding to the constructor/destructor. `Term` is the sequential term corresponding to the constructor/destructor branch of the fold/unfold terms.

B.22 RecordEntry

A `RecordEntry` has syntax given by:

```
RecordEntry ->  UIdent ":=" Term
              | UIdent ":=" [Ident] "->" Term.
```

B.23 Process

A `Process` is a list of process commands.

```
Process ->   "do" [ProcessCommand]
           | ProcessCommand.
```

B.24 ProcessCommand

A process command has syntax given by:

```
ProcessCommand ->  Ident "(" [Term] "|" [Channel] "=>" [Channel] ")"
                 | "close" Channel
                 | "halt" Channel
                 | "get" Ident "on" Channel
                 | "put" Term "on" Channel
                 | "hcase" Channel "of" [Handler]
                 | "hput" UIdent "on" Channel
                 | "split" Channel "into" [Channel]
                 | "fork" Channel "as" [ForkPart]
                 | "plug" [Process]
                 | Channel "|=|" PChannel

PChannel ->   Channel
            | "neg" Channel.
```

where:

- `Ident "(" [Term] "|" [Channel] "=>" [Channel] ")"` is the syntax for process call. `Ident` is the name of the process being called. `[Term]` is the list of sequential arguments, and `[Channel] "=>" [Channel]` represents the channels names (input and output), separated by `"=>"`, which are the concurrent arguments of the process call.

- `"close" Channel` is the syntax for closing channels.
- `"halt" Channel` is the syntax for halting channels.
- `"get" Ident "on" Channel` is the syntax for getting something on a channel. `Ident` is a variable name to which the term received on a channel binds to.
- `"put" Term "on" Channel` is the syntax for putting a term on a channel.
- `"hput" UIdent "on" Channel` is the syntax for putting a handle on a channel. The handle is represented by `UIdent`.
- `"hcase" Channel "of" [Handler]` represents hcasing on the (co)handles of a (co)protocol.
- `"split" Channel "into" [Channel]` represents splitting of a channel into two channels.
- `"fork" Channel "as" [ForkPart]` represents forking a process into two processes along a channel. The two processes are represented as a list of `ForkPart` which are arranged in a layout syntax with respect to the `"as"` keyword.
- `plug [Process]` represents plugging of multiple processes, represented by `PlugPart`, along channels. The different processes are arranged in a layout syntax with respect to the `"plug"` keyword.
- `Channel "|=|" PChannel` represents equating of two channels. The `Pchannel` is either a normal channel or the negation of a channel.

B.25 ForkPart

`ForkPart` has syntax given by:

```
ForkPart -> Ident Process.
```

where `Ident` is a channel on which the process is forked.

Appendix C

Abstract Syntax Tree for MPL

The *abstract syntax tree* (AST) for MPL, which is a Haskell data type, is provided in this appendix. We represent the data types of the nodes of the AST by setting up type aliases which keep the AST readable. AST* is a subset of AST in which the functions and processes are not type annotated. AST** is a subset of AST* in which there is no pattern-matching syntax.

An MPL program is a list of statements. A statement can either be a definition statement or a run statement. A definition statement can either be a normal definition or a `defn` construct with a `where` clause. A run statement represents the *main process* of an MPL program.

```
type PosnPair = (Int,Int)
type MPL      = [Stmt]

data Stmt =  DefnStmt ([Defn],[Stmt],PosnPair)
           | RunStmt (FunType,[PChannel],[PChannel],Process,PosnPair)
```

A definition can be a (co)data definition, (co)protocol definition, function definition or a process definition.

```
type Name       = String
type NumArgs    = Int
type Param      = String
type PChannel   = String

data DataName   = DataName (Name,[Param])
type ProtName   = DataName

type DataClause = (DataName,[DataPhrase])
type DataPhrase = (Nam,FunType,NumArgs)

type ProtocolClause = (ProtName,[ProtocolPhrase])
type ProtocolPhrase = (Name,FunType)

data Defn =
    Data([DataClause],PosnPair)
  | Codata([DataClause],PosnPair)
  | ProtocolDefn([ProtocolClause],PosnPair)
  | CoprotocolDefn([ProtocolClause],PosnPair)
  | FunctionDefn(FuncName,FunType,[(PatternTermPhr,PosnPair)],PosnPair)
  | ProcessDefn(Name,FunType,PattProcessPhr,PosnPair)
```

A function type and a function name is defined as:

```
data FunType =
    NoType | StrFType([String],Type) | IntFType([Int],Type)
```

```
data FuncName = Custom  Name | BuiltIn Func


data Func =
    Add_I | Sub_I | Mul_I | DivQ_I | DivR_I | Eq_I | Neq_I |
    Leq_I | Geq_I | LT_I  | GT_I | Eq_C | Eq_S | Concat_S |
    Unstring_S | ToStr | ToInt | Append| Or_B   | And_B
```

A `PatternTermPhr` forms the body of a function definition and is defined as:

```
type PatternTermPhr = ([Pattern],Either Term [GuardedTerm])
type GuardedTerm    = (Term,Term)


data Pattern =
    ConsPattern(String,[Pattern],PosnPair)
  | DestPattern(String,[Pattern],PosnPair)
  | ProdPattern([Pattern],PosnPair)
  | VarPattern(String,PosnPair)
  | DontCarePattern PosnPair
```

A sequential MPL term is represented as:

```
data Term =
      TRecord[(Pattern,Term,PosnPair)]
    | TCallFun(FuncName,[Term],PosnPair)
    | TLet(Term,[Defn],PosnPair)
    | TVar(String,PosnPair)
    | TConst(BaseVal,PosnPair)
    | TIf(Term,Term,Term,PosnPair)
```

```
    | TCase(Term,[PatternTermPhr],PosnPair)
    | TFold(Term,[FoldPattern],PosnPair)
    | TUnfold(Term,FoldPattern,PosnPair)
    | TCons(Name,[Term],PosnPair)
    | TDest(Name,[Term],PosnPair)
    | TProd([Term],PosnPair)
    | TDefault PosnPair

data BaseVal =
    ConstInt Int | ConstChar Char |
    ConstString String | ConstDouble Double

type FoldPattern = (Name,[Pattern],Term,PosnPair)
```

A `PattProcessPhr` represents the body of a process and consists of sequential pattterns, channels, and a list of process commands. It is represented as:

```
type PattProcessPhr = ([Pattern],[PChannel],[PChannel],Process)
type Process        = [ProcessCommand]

data ProcessCommand
    =  PRun(Name,[Term],[PChannel],[PChannel],PosnPair)
     | PClose(PChannel,PosnPair)
     | PHalt(PChannel,PosnPair)
     | PGet(Pattern,PChannel,PosnPair)
     | PPut(Term,PChannel,PosnPair)
     | PHPut(Name,PChannel,PosnPair)
     | PHCase(PChannel,[(Name,Process,PosnPair)],PosnPair)
```

```
      | PSplit(PChannel,(PChannel,PChannel),PosnPair)
      | PFork(String,[(PChannel,[PChannel],Process)],PosnPair)
      | PPlug([PChannel],(([PChannel],Process),([PChannel],Process)),PosnPair)
      | PCase(Term,[PattProc],PosnPair)
      | PId(Channel,Channel,PosnPair)
      | PNeg(Channel,Channel,PosnPair)

data Channel  =  PosChan PChannel
               | NegChan PChannel
               deriving (Eq,Show)
```

Types in MPL, both sequential and channel types are defined as:

```
type InputType  = [Type]
type OutputType = Type

data Type = Unit PosnPair
           |TypeDataType (String,[Type],PosnPair)
           |TypeCodataType(String,[Type],PosnPair)
           |TypeProd([Type],PosnPair)
           |TypeConst(BaseType,PosnPair)
           |TypeVar(String,PosnPair)
           |TypeVarInt Int
           |TypeFun([Type],Type,PosnPair)
           |Get(Type,Type,PosnPair)
           |Put(Type,Type,PosnPair)
           |Neg(Type,PosnPair)
           |TopBot PosnPair
```

```
            |ProtNamed(String,[Type],PosnPair)
            |CoProtNamed(String,[Type],PosnPair)
            |ProtTensor(Type,Type,PosnPair)
            |ProtPar(Type,Type,PosnPair)
            |ProtProc([Type],[Type],[Type],PosnPair)

data BaseType =  BaseInt
               | BaseChar
               | BaseDouble
               | BaseString
```

Appendix D

Data Type for the Symbol Table of MPL

A symbol table is a list of `ScopeSymbols`, where `ScopeSymbols` represent a list of `SymbolDefn`. A `SymbolDefn` is a data type that stores the information about various MPL definitions, and consists of one node for every MPL definition:

```
type SymbolTable  = [ScopeSymbols]
type ScopeSymbols = [SymbolDefn]
```

A `SymbolDefn` is defined as:

```
type NumArgs  = Int
type Name     = String
type Param    = String


data DataName = DataName (Name,[Param])
data FunType  =   NoType
                | StrFType([String],Type)
                | IntFType([Int],Type)
```

```
-- ((data Name,[constructor names]),constructor type,fold Type of
-- constructor,number of constructor args)
type ConsVal = ((DataName,[Name]),FunType,(FunType,FunType),NumArgs)
type DestVal = ((DataName,[Name]),FunType,(FunType,FunType),NumArgs)
type HandVal = ((ProtName,[Name]),FunType,NumArgs)

-- this is the data type inserted in the symbol table
data SymbolDefn  =
      SymData[(Name,ConsVal)]
    | SymCodata[(Name,DestVal)]
    | SymProt[(Name,HandVal)]
    | SymCoProt[(Name,HandVal)]
    | SymFun[(FuncName,(FunType,NumArgs))]
    | SymProc[(Name,(FunType,(NumArgs,NumArgs,NumArgs)))]
```

`ValLookup` is the data type that can be looked up from the symbol table:

```
data ValLookup =
      Val_Cons(Name,PosnPair)
    | Val_Dest(Name,PosnPair)
    | Val_Prot(Name,PosnPair)
    | Val_Coprot(Name,PosnPair)
    | Val_TypeSyn(Name,PosnPair)
    | Val_Fun(FuncName,PosnPair)
    | Val_Proc(Name,PosnPair)
```

`ValRet` is the data type returned as a result of lookup from the symbol table:

```
data ValRet =
```

```
  ValRet_Cons((DataName,[Name]),FunType,(FunType,FunType),NumArgs)
| ValRet_Dest((DataName,[Name]),FunType,(FunType,FunType),NumArgs)
| ValRet_Prot(ProtName,FunType,NumArgs)
| ValRet_Coprot(ProtName,FunType,NumArgs)
| ValRet_Fun(FunType,NumArgs)
| ValRet_Proc(FunType,(NumArgs,NumArgs,NumArgs))
```

Appendix E

Data Type for Core MPL

Core MPL is an intermediate language of MPL. There is no pattern-matching syntax or local functions in Core MPL. The Haskell data type for Core MPL is given in this appendix.

An MPL program consists of various definitions:

```
type Include_Defns    = Defn
type Data_Defns       = [Defn]
type Codata_Defns     = [Defn]
type Protocol_Defns   = [Defn]
type CoProtocol_Defns = [Defn]
type Function_Defns   = [Defn]
type Process_Defns    = [Defn]
type MainRun_Defn     = Defn

data MPLProg  = MPLProg  Include_Defns Data_Defns  Codata_Defns
                         Protocol_Defns CoProtocol_Defns
                         Function_Defns Process_Defns  MainRun_Defn
```

An MPL definition is:

```
type Name       = String
type PosnPair   = (Int,Int)
type NamePnPair = (Name,PosnPair)
type Argument   = (String,PosnPair)
type Channel    = NamePnPair

-- data name and a list of constructors
type DataClause  = (NamePnPair,[Constructor])

-- constructor name and number of argument it takes
type Constructor = (NamePnPair,(Int,PosnPair))

-- protocol name & list of handles
type ProtocolClause  = (NamePnPair,[NamePnPair])

data Defn  =
      Includes [String]
    | Data(PosnPair,[DataClause])
    | Codata(PosnPair,[DataClause])
    | Protocol(PosnPair,ProtocolClause)
    | CoProtocol(PosnPair,ProtocolClause)
    | Function(PosnPair,FuncName,[Argument],Term) |
    | Process(PosnPair,NamePnPair,[Argument],
              [Channel],[Channel],[ProcessCommand])
    | MainRun(PosnPair,[Channel],[Channel],[ProcessCommand])
```

A sequential term is defined as:

```
type Struct_Name   = (NamePnPair,NamePnPair)
type PatternDef    = (Struct_Handle ,[Term] )
type Struct_Handle = (NamePnPair,NamePnPair,[NamePnPair])


data Term =
      TCall(FuncName,[Term])
    | TCons(Struct_Name,[Term])
    | TDest(NamePnPair,Struct_Name,[Term])
    | TCase(Term,[PatternDef],PosnPair)
    | TVar(String,PosnPair)
    | TConstS(String,PosnPair)
    | TConstC(Char,PosnPair)
    | TConstI(Int,PosnPair) |
    | TRec([(Struct_Handle,Term)],PosnPair)
    | TProd [Term]
    | TProdElem(Int,Term,PosnPair)
```

A function can either be a custom or built-in function. It is defined as:

```
data FuncName =   Custom(String,PosnPair)
                | Inbuilt(Func,PosnPair)


data Func =
   Add_I | Sub_I | Mul_I  | DivQ_I | DivR_I |
```

```
    Eq_I  | Leq_I | Eq_C  | Leq_C  | Eq_S   | Leq_S  |
    Concat_S Int | Unstring_S | ToStr | ToInt |
    Append | Or_B | And_B
```

A process command is defined as:

```
type Event_Handle   = (NamePnPair,NamePnPair)
type Struct_Handle  = (NamePnPair,NamePnPair,[NamePnPair])

type ProcessPhrase_hcase  = (Event_Handle,Process)
type ProcessPhrase_pcase  = (Struct_Handle,Process)

type ForkPart = (Channel,[Channel],Process)
type Process  = [ProcessCommand]

data ProcessCommand =
       PRun(PosnPair,NamePnPair,[Term],[Channel],[Channel])
     | PClose(PosnPair,Channel)
     | PHalt(PosnPair,[Channel])
     | PGet(PosnPair,NamePnPair,Channel)
     | PHcase(PosnPair,Channel,[ProcessPhrase_hcase ])
     | PPut(PosnPair,Term ,Channel)
     | PHput(PosnPair,Event_Handle,Channel)
     | PSplit(PosnPair,Channel,[Channel])
     | PPlug(PosnPair,[Channel],([Channel],Process),([Channel],Process))
     | PFork(PosnPair,Channel,[ForkPart])
     | PCase(PosnPair,Term ,[ProcessPhrase_pcase])
```

```
      | PRec(PosnPair,[(Struct_Handle,[ProcessCommand])])
      | PId (PosnPair,PChannel,PChannel)

data PChannel = PosChannel String
              | NegChannel String
```

Appendix F

Data Type for the Abstract Machine of MPL

This appendix describes the Haskell data type for the abstract machine for MPL (AMPL).

AMPL code consists of a list of various AMPL definitions: (co)data, (co)protocol, function, and process. It is defined as:

```
type Name       = String
type PosnPair   = (Int,Int)
type NamePnPair = (Name,PosnPair)
type Channel    = String
type Argument   = String


-- (co)protocol definition
data HANDLE_SPEC = Handle_spec NamePnPair [Name]


-- (co)data definiton
data STRUCTOR_SPEC = Struct_spec NamePnPair [(Name,Int)]
```

```
-- process definition
data PROCESS_SPEC
   = Process_specf NamePnPair [Argument] ([Channel],[Channel]) COMS

-- function definition
data FUNCTION_SPEC = Function_spec NamePnPair [Argument] COMS

data AMPLCODE
    = AMPLcode [HANDLE_SPEC] [HANDLE_SPEC] [STRUCTOR_SPEC]
               [STRUCTOR_SPEC] [PROCESS_SPEC] [FUNCTION_SPEC]
               (PosnPair,[Channel],[Channel],COM])
```

An AMPL command is defined as:

```
type STRUCTOR_NAME = (Name,Name)

data COM =
   AC_ASSIGN String COM | AC_STORE String | AC_LOAD String | AC_RET |
   AC_CALL NamePnPair [Argument] | AC_STRING String | AC_EQS | AC_LEQS |
   AC_CONCAT Int | AC_UNSTRING | AC_CHAR Char | AC_EQC | AC_LEQC |
   AC_INT Int | AC_LEQ | AC_EQ | AC_ADD | AC_SUB | AC_MUL | AC_DIVQ |
   AC_DIVR | AC_TOSTR  | AC_TOINT | AC_OR | AC_AND | AC_APPEND |
   AC_STRUCT STRUCTOR_NAME [NamePnPair] |
   AC_CASE [(STRUCTOR_NAME,[NamePnPair],[COM])] |
   AC_GET String Channel |
   AC_PUT String Channel |
   AC_HPUT String STRUCTOR_NAME |
   AC_SPLIT Channel (Channel,Channel) |
```

```
AC_FORK Channel ((Channel,[Channel],[COM]),(Channel,[Channel],[COM])) |
AC_PLUG [Channel] ([Channel],[COM]) ([Channel],[COM]) |
AC_ID Channel Channel | AC_CLOSE Channel | AC_HALT [Channel] |
AC_HCASE Channel [(STRUCTOR_NAME,[COM])] |
AC_RUN NamePnPair [Argument] ([Channel],[Channel]) |
AC_RECORD [(STRUCTOR_NAME,[Name],[COMS])] | AC_PROD [Argument]
AC_PRODELEM Int String
```