UNIVERSITY OF CALGARY

The Type System for the Message Passing Language MPL

by

Subashis Chakraborty

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JANUARY, 2014

Abstract

We present a formal description of the Message Passing Language (MPL) which is based on the message passing logic proposed by Cockett and Pastro with the addition of concurrent datatypes or protocols. This thesis also provides a "proof of concept" of the language by developing substantial examples in MPL. We also present a complete type system for MPL which underpins both the type checking and the type inference of the language.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor, Dr. Robin Cockett, for giving me enormous support and advice to complete this thesis. It has been a really great experience to work under his supervision.

I would like to thank Kristine Bauer, Robin Cockett, Philip W. L. Fong, and Lionel Vaux, for serving on my thesis committee.

I am grateful to Brett Giles, Sutapa Dey, Tannistha Maiti, Sumaya Habib and Didar-Al-Alam for proof reading this thesis and giving me valuable suggestions. Any remaining errors are mine.

I would like to thank all of my friends, relatives and labmates for their support.

I would like to take this opportunity to thank Jonathan Gallagher for his advice and valuable suggestions during the development of this thesis.

A special thanks to my wife Masuka Yeasin for her patience, love and also for proof reading. I would also like to thank my dada, boudi and sweet Suchismita and Soumik for their love and inspiration. Finally, special thanks to my wonderful parents for their support, inspiration, and love.

Dedicated to my parents.

Table of Contents

List of Figures and Illustrations

Chapter 1

Introduction

This thesis introduces the concurrent programming language MPL (Message Passing Language) which is based on an extension of linear logic [**?**] which allows messages from a sequential world to be passed between processes [**?**]. The thesis explains the language and provides examples of programs in MPL. However, its main technical contribution is to specify the type system for MPL. The type system underpins both the type checking and the type inference of the language.

MPL is based on a two-tier logic for message passing proposed by Cockett and Pastro, in [**?**]. The logic serves as a proof theory for concurrency in which livelocks and deadlocks are impossible. The first tier of the logic is just a proof theory for sequential programs (such as the λ-calculus) which we refer to as the "sequential world". The second tier, which is built on top of the first tier, is a proof theory of concurrent programs which uses message passing primitives to pass sequential data. We refer to it as the "concurrent world". MPL maintains a strict separation between sequential computations (sequential world) and interacting processes (concurrent world). The correspondence between the proof theory, the categorical semantics, and the internal language for this two-tier logic was established in [**?**]. The internal language of the categorical semantics gives a basic programming language for concurrent programs in which processes communicate by passing messages via channels.

The categorical semantics of concurrent datatypes was introduced in [**?**]. Types for interaction in the concurrent world are called protocols. Concurrent datatypes in MPL are, thus, user defined protocols. A protocol determines the legal actions on a channel: such actions consist of sending and receiving messages on the channel and creating new channels

to connect to processes which are spawned during interaction. The addition of user defined protocols to MPL, makes the language much more useable as will be demonstrated in this thesis. The programs in MPL determine how processes interact and use sequential data.

1.1 Organization

This thesis defines the type system for MPL: it is divided into the sequential type system and the concurrent type system. The type system, as it is based in the message passing logic [**?**], ensures that the well typed programs of MPL have no livelock, deadlock, or runtime errors.

The thesis is organized as follows: Chapter 2 provides an overview of MPL programs. This chapter also describes the use of layout in MPL programs. Chapter 3 presents an introduction to sequential MPL and Chapter 4 presents a description of the concurrent part of MPL. In Chapter 5, some more substantial examples in MPL are developed to illustrate the use of the language. Chapter 6 is devoted to giving a formal type system for the sequential part of MPL. Chapter 7 is devoted to giving a formal type system for the concurrent part of MPL. Chapter 8 presents conclusion and possible future work.

1.2 Contribution of this thesis

Chapter 3 and 4 provides the description of the Message Passing Language (MPL). The thesis then explores examples of programs to illustrate programming in MPL. Chapter 5 presents more substantial MPL programs which further explore expressiveness of MPL.

The main contribution of the thesis is a complete description of the type system of the language. This is broken into two chapters, namely Chapter 6 and Chapter 7. Chapter 6 gives the type system for the sequetial world while Chapter 7 gives the type system for the

concurrent world. These type systems are the basis for type checking and type inference in MPL.

A type inference program was originally developed by Sean Nichols for MPL. However, there was little documentation of the system which, in addition, contained a number of "bugs" (which have now been fixed). This thesis contains a more complete description of MPL and in particular the formal description of its type system which underpins that type inference algorithm.

1.3 Related work

1.3.1 Types in general

Types in a programming language are of great theoretical and practical importance. Typed programming languages eliminate the source of many programming errors at compile time. For example, if $f : \mathsf{Z} \to \mathsf{Z}$ and $l : [Z]$ then $f(l)$ will not type check and this will be detected as an error at compile time. Typed programming encourages a "compositional" development style, and the type checker can be used to ensure that the composition of a program out of smaller modular components fits together correctly.

Type systems have many applications: they have been used to give programs running on the Linux operating system a memory safe execution environment [**?**]. In the DecaC programming language, types are used to prevent certain common pointer errors such as dangling pointers and double-free errors. In the Applied Type System (ATS) language [**?**], types are used to ensure a completely safe use of pointers. ATS also uses type information to optimize the code it generates and produces binaries that run as fast or faster than C.

1.3.2 Formal programming

One of the interesting features of the simply typed λ-calculus (with a natural number object see [**?**]), which is a basic formal programming language, is that all computations are total and terminate (in the sense that every term reduces to a head normal form). Formal languages which guarantee termination in this sense cannot be Turing complete. So, for example, one cannot write an interpreter for the language in the language itself. However, all feasible computations can be expressed (although they may require much longer and less efficient programs). The typed λ-calculus can be turned into a much more expressive programming language by adding inductive and coinductive datatypes. The addition of coinductive data, in particular, allows one to program with potentially infinite data and adds an expressive aspect not present in most traditional languages. Charity is a programming language based on these ideas [**?**, **?**]: it is a typed programming language with inductive and coinductive data, and moreover, to ensure termination, the only source of recursion and corecursion is through the universal properties of that data. In [**?**], Turner argues that a "total" functional programming language (by which he meant a language using Walther recursion [**?**] which is very similar to the form of primitive recursion used above in Charity), is powerful enough for "production programming".

These formal programming systems are common components of proof assistants. For example Agda [**?**, **?**] is an extension of Martin-Löf type theory [**?**] and implements a dependent type system which has inductive data. Another similar and well-known system with dependent types is Coq [**?**] which is an extension of the Calculus of Constructions [**?**]: it also has inductive data.

The Message Passing Language (MPL) that we are going to describe in this thesis, has initial and final data with their respective recursion and corecursion combinators following the approach used in Charity [**?**, **?**].

1.3.3 Formal concurrent programming

In [?], Milner introduced the π-calculus which is based on the notion of message passing along channels between processes. Milner's notion of process calculus allows many processes to communicate on one channel so the π-calculus is highly non-deterministic. In MPL, a deterministic approach is taken to concurrent processes: in particular, a channel only connects exactly two processes.

In [?], [?], a notion, called session types, was introduced to guide the communication between two processes in a concurrent program. In [?], session types are described to implement concurrent communication for a functional programming language and in [?], Sackman and Eisenbach, incorporate session types into Haskell as a tool for concurrent programming.

In recent work [?], Wadler describes how propositions of classical linear logic correspond to session types. These session types can be seen as the user defined protocols (in the sense of this thesis) which describe the sequence of allowable actions on a channel which connects two processes.

Chapter 2

Introduction to MPL

This chapter provides an overview of MPL (Message Passing Language). MPL is a concurrent language based on Cockett and Pastro's message passing logic [**?**] with the addition of datatypes or protocols to it. A central feature of this language is the separation between sequential computations (sequential world) and interacting processes (concurrent world); MPL programs describe how processes interact and use sequential computations. MPL provides a strong type system for programs and processes. In particular, a well typed program has no livelock or deadlock, and has no runtime errors.

The grammar for MPL programs is given in Appendix A. An MPL program consists of a sequence of definitions followed by the `run` command which executes a process. Each definition is either a data declaration, a protocol declaration, a function definition, or a process definition. The `run` command initiates the execution of an MPL program by specifying the process at which the program will start its execution.

2.1 Comments

MPL uses Haskell-style comments [**?**]: there are multi-line comments:

{−

comment

−}

and single-line comments

−− *comment.*

Comments are removed in the first preprocessing step before tokenizing.

2.2 Layout

A program written in MPL depends on its layout. While, one is not forced to use layout, it is more convenient to do so. The rules governing the layout of MPL programs are provided in Appendix B. Layout uses the offside rule described by Peter J. Landin, in [**?**] as follows: "Any non-whitespace token to the left of the first such token on the previous line is taken to be the start of a new declaration." The layout preprocessing causes explicit tokens, called off-side markers to be added to the program and relieves the programmer from having to use explicit separators. Consider the following program written in MPL:

```
1 data Bool -> c = True : c
2                  False : c
3 or : Bool -> Bool -> Bool
4 or x y =
5    case x of
6         True  -> True
7         False -> y
```

Note how layout is used to group the `True` and `False` constructors in the data declaration in line 1 and 2 and also in line 6 and 7. Here no explicit semicolon is required as same indentation level is maintained in both cases and the layout preprocessing step will add the explicit off-side markers required.

The same program with explicit semicolon separators looks like:

```
1 data Bool -> c = True : c; False : c
2 or : Bool -> Bool -> Bool; or x y =
3        case x of
4             True  -> True; False -> y
```

Note that, after "=" and "of", no explicit semicolon is required, because these are special keywords which reset the indentation level to that of the next token.

2.3 Sequential programs

The sequential part of MPL is functional in style. Consider the example, in Figure 2.1, of a sequential program to compute the length of a list whose values are of arbitrary type. To compute the length of a list it is necessary to recurse over the list: MPL only allows disciplined recursion using "folding" (or catamorphisms), hence the program is introduced with the `fold` command: this is an example of a fold function definition.

In line 9, the name of the function, `length` and type of the function `List a` $\rightarrow$ `Nat` are given separated by ":". Thus, `length` has type `List a` $\rightarrow$ `Nat` which means, explicitly, the function `length` takes a list of arbitrary type as input and produces the length of the list as a unary natural number. The types `List a` and `Nat` are defined in line 1 to 5 in Figure 2.1 using data declarations. The data declarations and the recursive definition using the `fold` construct are all sequential constructs of the MPL language. The sequential world of MPL is discussed further in chapter 3.

Notice, however, to actually run a sequential program one must create a process which uses the sequential program. Thus to run a MPL program a process which is a concurrent term in MPL must be defined. From line 14-16, the process, `Test`, is defined which applies `length` to a list provided by the "user". The process arranges the interaction with user. Notice, the `Test` process has one input (polarity) channel, `user`, but no output (polarity) channels. These input and output channels are always separated by $\Rightarrow$. In line 19, the `run` keyword followed by the process, `Test`, executes the process.

2.4 Concurrent programs

A process always has one or more channels associated to it. Thus, processes can be easily identified by the fact that they always have channels associated to them which are separated

```
1  -- data definitions
2  data Nat -> c = Zero : c
3                  Succ : c -> c
4  data List a -> c = Nil : c
5                     Cons : a -> c -> c
6
7  -- length function given by fold combinator
8  fold
9    length : List a -> Nat
10   length x by x =
11        Nil       -> Zero
12        Cons y ys -> Succ (length ys)
13
14 -- process needed to run the computation in length function
15 Test user => =
16    get x on user. put (length x) on user; end user
17
18 -- initializing command to start the execution of a MPL program
19 run Test
```

Figure 2.1: Sequential program in MPL

by ⇒ into input and ouput polarity channels. As channels in MPL allow two-way communication, putting a channel in "input" position does not mean the channel only provides inputs to the process; similarly, putting a channel in "output" position does not mean the channel only provides outputs. A channel can potentially recieve and send messages whether it is an "input" or "output" channel in this sense.

Thus while processes have input polarity and ouput polarity channels, because channels are two-way communication devices, this does not determine the direction in which messages are passed. Placing a channel to the left of ⇒ (i.e. giving it an input polarity) determines only that recieving and sending messages will be seen from the perspective of the external world. While placing a channel to the right of ⇒ (i.e. giving it an output polarity) means these actions are seen from the perspective of the process itself. Thus the "polarity" of the

channel is the perspective from which it is to be viewed. When plugging processes together the channels must have opposite polarities but be of the same type.

An example of a concurrent program in which two processes are plugged together is given in Figure 2.2. A simple process, `TwoWayTalk` is given in line 5-17. It uses the `drive` construct which allows recursive definitions in the concurrent world, analogous to the `fold` in the sequential world. For a fuller description see Chapter 4. This process receives messages on each of its output channels, namely `y` and `z`, and passes these messages as a pair on its input channel, `x`. Then the process takes a message on its input channel, `x`, and sends a copy of the message on each of the output channels, `y` and `z`. The type of a channel or protocol is determined by the actions possible on a channel: the protocol, in this case, is defined in line 1-3. We wish to plug this process to another process which we describe next.

The process, `Average` has one input channel `w` and one output channel `x`. It recieves a pair of numbers on channel `x`, and then sends the average on channel `w`: the `average` function is assumed to have been defined elsewhere. The process `Average` then recieves a message on `w` and pass it on to `x`.

As the input channel of `TwoWayTalk` and the output channel of `Average` have the same protocol, these two processes can be plugged together to make a compsite process `TwoAverage`. In line 34-37, `plug` command is used to plug `Average` to `TwoWayTalk` on channel `x`.

Finally one runs the process `TwoAverage`.

```
1  -- protocol definition
2  protocol Talk (a b) => $C =
3         #response :: get a (put b $C) => $C
4
5  -- process definitions
6  drive
7    TwoWayTalk :: () Talk ((a * b) b) => Talk (a c), Talk (b c)
8    TwoWayTalk x => y, z by x =
9          #response: #response on y
10                    #response on z
11                    get a1 on y.
12                    get a2 on z.
13                    put (a1 * a2) on x
14                    get b on x.
15                    put b on y
16                    put b on z
17                    call TwoWayTalk x => y, z
18
19 drive
20   Average :: () Talk (a b) => Talk ((a * b) c)
21   Average w => x by w =
22              #response: #response on x;
23                         get a on x.
24                         case a of
25                              (a1 * a2) -> put (average a1 a2) on w
26                                           get a on w.
27                                           put a on x
28                                           call Average w => x
29
30 -- This process is a composite process via
31 -- plugging of two above processes
32 TwoAverage :: () Talk ((a * b) c) => Talk (a b), Talk (a b)
33 TwoAverage w => y, z  =
34           plug on x
35                call Average w => x
36                to
37                call TwoWayTalk x => y, z
38
39 -- initializing command to start the execution of a MPL program
40 run TwoAverage
```

Figure 2.2: Concurrent program in MPL

Chapter 3

Introduction to Sequential world

The sequential part of MPL consists of data definitions, their operations, basic term constructions, and function definitions. MPL separates the data definitions into two classes: the inductive data definitions (`data`) and the coinductive data definitions (`codata`).

3.1 Inductive data

The general form of an inductive data declaration is:

$$\begin{aligned} \texttt{data D } \widetilde{\texttt{P}} \rightarrow \texttt{z} = \texttt{c}_1 : \texttt{F}_1 \ \widetilde{\texttt{P}_1} \ \texttt{z} \rightarrow \texttt{z} \\ \vdots \\ \texttt{c}_\texttt{n} : \texttt{F}_\texttt{n} \ \widetilde{\texttt{P}_\texttt{n}} \ \texttt{z} \rightarrow \texttt{z} \end{aligned}$$

where,

1. `D` is the name of the datatype.
2. $\widetilde{\texttt{P}}$ is a sequence of type parameters such as $\texttt{p}_1 \cdots \texttt{p}_\texttt{m}$ and $\widetilde{\texttt{P}_\texttt{i}}$ is a sequence of type parameters for $\texttt{c}_\texttt{i}$.
3. $\texttt{c}_\texttt{i}$ is the name of the constructor where `i` ranges from `1` to `n`.
4. `z` is the state variable.
5. $\texttt{F}_\texttt{i} \ \widetilde{\texttt{P}_\texttt{i}} \ \texttt{z}$ is a type using the type varibales $\widetilde{\texttt{P}_\texttt{i}}$ and `z`.

Inductive data is a least fixed point, $\mu z.F(z)$, of a functor, F. The state variable, z of an inductive data declaration is the bound variable in the fixed point definition, i.e. the z in $\mu z.F(z)$.

Basic examples of inductive data definition are the boolean type, the maybe type, the natural numbers, and lists of a given type:

```
-- boolean type definition
data Bool -> c = True : c
                 False : c
-- maybe type definition for representing possible failure
data Maybe a -> c = Just : a -> c
                    Nothing : c
--natural number type definition
data Nat -> c = Zero : c
                Succ : c -> c
-- list type definition
data List a -> c = Nil : c
                   Cons : a -> c -> c
```

3.1.1 Constructors

An inductive data declaration has a set of constructors. For example, the boolean data declaration, `Bool`, has two constructors: `True` and `False`. The `Maybe` type also has two constructors, Just and Nothing.

The natural number type, `Nat` has two constructors: `Zero` and `Succ`. To obtain the types of these constructors, one substitutes the state variable `c` by `Nat`: thus `Zero` has the type `Nat`, and `Succ` has the type `Nat` $\to$ `Nat`. The natural number type is recursive and this allows infinitely many terms of `Nat` to be constructed:

$$\texttt{Zero},\ \texttt{Succ}(\texttt{Zero}),\ \texttt{Succ}(\texttt{Succ}(\texttt{Zero})),\ \cdots.$$

These are often written as $0,\ 1,\ 2,\ \cdots$.

The list type, `List` has two constructors: `Nil` and `Cons`. To obtain the types of these constructors, one substitutes the state variable `c` by `List a`: thus `Nil` has the type `List a`, and `Cons` has the type `a` $\to$ `List a` $\to$ `List a`. This allows, for example, the following terms

to be constructed:

$$\texttt{Nil},\ \texttt{Cons 1 Nil},\ \texttt{Cons 2 (Cons 1 Nil)}.$$

These are often written as [ ], [1], [2, 1] respectively.

3.1.2 Case

The `case` construct allows programs using inductive data to be conditional on the outermost constructor. The `case` construction has the following general form:

$$\begin{array}{l}\texttt{case t of}\\ \quad \texttt{c}_1\ \texttt{x}_{11}\cdots\texttt{x}_{1\texttt{m}_1} \to \texttt{t}_1\\ \quad \cdots\\ \quad \texttt{c}_\texttt{n}\ \texttt{x}_{\texttt{n}1}\cdots\texttt{x}_{\texttt{nm}_\texttt{n}} \to \texttt{t}_\texttt{n}\end{array}$$

where $\texttt{t}$ and $\texttt{t}_1, \cdots, \texttt{t}_\texttt{n}$ are terms such that the variables $\texttt{x}_{\texttt{i}1}, \cdots, \texttt{x}_{\texttt{im}_\texttt{i}}$ are bound in $\texttt{t}_\texttt{i}$.

In Figure 3.1, the `case` is used to write the predecessor function on natural numbers.

```
1 pred : Nat -> Nat
2 pred a =
3     case a of
4          Zero ->   Zero
5          Succ n -> n
```

Figure 3.1: MPL function for calculating the predecessor of a natural number

Data like `Bool` and `Maybe a` are not recursive: thus, the `case` construct is the main operation for these types. The `case` can be used to express the boolean operations `and` and `or`, see Figure, 3.2.

3.1.3 Fold

The `fold` combinator is the basic recursion operation in sequential MPL, and is provided automatically once inductive data is declared. The `fold` is the only way to recursively

```
1  -- and operation on booleans
2  and : Bool -> Bool -> Bool
3  and x y =
4     case x of
5          True  -> y
6          False -> False
7  -- or operation on booleans
8  or : Bool -> Bool -> Bool
9  or x y =
10      case x of
11           True  -> True
12           False -> y
```

Figure 3.2: The `and` and `or` operations on booleans by using case

process recursive data. The `fold` is referred to as a "catamorphism" by Meijer, et.al in [**?**, **?**] and provides primitive recursion on inductive data.

The `fold` has the following general form:

$$
\begin{array}{l}
\texttt{fold} \\
\quad \texttt{f} : \texttt{T} \\
\quad \texttt{f}\ \texttt{x}_1\ \cdots\ \texttt{x}_\texttt{p}\ \texttt{by}\ \texttt{x}_\texttt{k} = \\
\qquad \texttt{c}_1\ \texttt{y}_{11}\ \cdots\ \texttt{y}_{1\texttt{q}_1}\ \rightarrow\ \texttt{t}_1 \\
\qquad\qquad \vdots \\
\qquad \texttt{c}_\texttt{m}\ \texttt{y}_{\texttt{m}1}\ \cdots\ \texttt{y}_{\texttt{m}\texttt{q}_\texttt{m}}\ \rightarrow\ \texttt{t}_\texttt{m}
\end{array}
$$

where,

- `T` is the type of the function `f`.
- $\texttt{x}_1 \cdots \texttt{x}_\texttt{p}$ are the names of the parameters of the function `f` and at the parameter $\texttt{x}_\texttt{k}$ on which the recursion is performed.
- $\texttt{c}_1 \cdots \texttt{c}_\texttt{m}$ are constructors.

- $\mathtt{t_1} \cdots \mathtt{t_m}$ are terms.
- $\mathtt{y_{i1}} \cdots \mathtt{y_{iq_i}}$ are the parameters of the constructors.

such that,

- `f` can occur in $\mathtt{t_1} \cdots \mathtt{t_m}$ (so `f` is in the scope of $\mathtt{t_1} \cdots \mathtt{t_m}$).
- The variables $\mathtt{y_{i1}} \cdots \mathtt{y_{iq_i}}$ are bound in $\mathtt{t_i}$.
- The type of the function `f` and the type of the terms $\mathtt{t_1} \cdots \mathtt{t_m}$ are same.
- This `fold` operation delivers a function `f` with type `T`.

The example in Figure 3.3, defines the `append` function using `fold`, which concatenates two lists. The `append` function recurses on the first argument, `xs` which is a list and is called recursively in line 6.

```
1 -- appending two lists
2 fold
3  append : List a -> List a -> List a
4  append xs ys by xs =
5       Nil       -> ys
6       Cons x xs -> Cons x (append xs ys)
```

Figure 3.3: MPL function for appending two lists

3.1.4 Mutual recursion on inductive data

One can write mutually recursive folds on non-mutually recursive inductive data in MPL. For mutual recursion, MPL allows many fold phrases under the keyword, `fold`. An example is given in Figure 3.4: this example determines whether a list is of even or odd length. The multiple fold phrases are used to define `even` and `odd` in a mutually recursive way.

```
1 -- determines even or odd number of elements in a list
2 fold
3    evens : List a -> Bool
4    evens xs by xs =
5          Nil       -> True
6          Cons e es -> Cons e (odds es)
7    odds : List a -> Bool
8    odds xs by xs =
9         Nil       -> False
10        Cons o os -> evens os
```

Figure 3.4: MPL functions for showing mutual recursion using a mutually recursive fold

One can also have mutually recursive inductive data in MPL. For mutually recursive data, the `fold` must be mutually recursive. In MPL, the `data` keyword sets an environment in which mutually recursive data can be declared, separated by the "and" keyword which allows sharing state variables. The general form of a declaration of mutually recursive inductive data is:

$$\begin{aligned}
\texttt{data}\ \mathtt{D_1}\ \widetilde{\mathtt{P}} \to \mathtt{z_1} = \mathtt{c_{11}} : \mathtt{F_{11}}\ \widetilde{\mathtt{P_{11}}}\ \mathtt{z_1} \to \mathtt{z_1} \\
\vdots \\
\mathtt{c_{1m}} : \mathtt{F_{1m}}\ \widetilde{\mathtt{P_{1m}}}\ \mathtt{z_1} \to \mathtt{z_1} \\
\vdots \\
\texttt{and}\ \mathtt{D_n}\ \widetilde{\mathtt{P}} \to \mathtt{z_n} = \mathtt{c_{nm}} : \mathtt{F_{n1}}\ \widetilde{\mathtt{P_{n1}}}\ \mathtt{z_n} \to \mathtt{z_n} \\
\vdots \\
\mathtt{c_{nm}} : \mathtt{F_{nm}}\ \widetilde{\mathtt{P_{nm}}}\ \mathtt{z_n} \to \mathtt{z_n}
\end{aligned}$$

An example showing the use of mutual recursion on mutually inductive data is given in Figure 3.5, which calculates the free variables of a predicate formula. The inductive data for predicate formula, is defined in line 1-7. It has a term construct which is a conditional on a predicate and this means the `Term` and `Prop` data declaration are defined in terms of

each other. One can substitute the state variables, `c` by `Term` and `d` by `Prop`, to obtain the types of the constructors. For example, `Var` has type `String` → `Term` and `Pr` has type `String` → `List Term` → `Prop`.

Then `fvTerm` and `fvProp` respectively calculates the free variables of terms and propositions where `union`, `remove` and `foldUnion` functions are assumed to have been defined elsewhere.

```
1  data Term -> c = Var : String -> c
2                   Cond : d -> c -> c -> c
3  and Prop -> d = Pr : String -> List c -> d
4                  And: d -> d -> d
5                  Or :  d -> d -> d
6                  Forall : String -> d -> d
7                  Exist : String -> d -> d
9  fold
10    map : (a -> b) -> (List a) -> List b
11    map f x by x =
12        Nil       -> Nil
13        Cons y ys -> Cons (f y) (map ys)
21 --calculating the list of free variables from terms and propositions
22 fold
23    fvTerm : Term -> List a
24    fvTerm x by x =
25        Var t ->  Cons t Nil
26        Tm t ts -> Cons t (foldUnion (map fvTerm ts))
27        Cond p t1 t2 -> union (propt p) (union (fvTerm t1) (fvTerm t2))
28    fvProp : Prop -> List a
29    fvProp y by y as
30        Pr p ts -> Cons p (foldUnion (map fvTerm ts))
31        And p1 p2 -> union (fvProp p1) (fvProp p2)
32        Or p3 p4 -> union (fvProp p3) (fvProp p4)
33        Forall p p1 -> remove p (fvProp p1)
34        Exist  p p1 -> remove p (fvProp p1)
```

Figure 3.5: MPL function for calculating the list of free variables from terms and propositions

3.2 Coinductive data

Dual to inductive data is coinductive data. The general form of a coinductive data declaration is:

$$\texttt{codata z} \rightarrow \texttt{D}\ \widetilde{\texttt{P}} = \texttt{d}_1 : \texttt{z} \rightarrow \texttt{F}_1\ \widetilde{\texttt{P}_1}\ \texttt{z}$$
$$\vdots$$
$$\texttt{d}_\texttt{n} : \texttt{z} \rightarrow \texttt{F}_\texttt{n}\ \widetilde{\texttt{P}_\texttt{n}}\ \texttt{z}$$

where,

1. D is the name of the codatatype.
2. $\widetilde{\texttt{P}}$ is a sequence of type parameters such as $\texttt{p}_1 \cdots \texttt{p}_\texttt{n}$ and $\widetilde{\texttt{P}_\texttt{i}}$ is a sequence of type parameters for $\texttt{c}_\texttt{i}$.
3. $\texttt{d}_\texttt{i}$ is the name of the destructor where i ranges from 1 to n.
4. z is the state variable.
5. $\texttt{F}_\texttt{i}\ \widetilde{\texttt{P}_\texttt{i}}\ \texttt{z}$ is a type using the type variables $\widetilde{\texttt{P}_\texttt{i}}$ and z.

Some examples of coinductive datatypes are triples, infinite lists, colists, cotrees etc. The datatype of triples is produced by the following definition:

```
codata t -> Triple a b c = First : t -> a
                         Second : t -> b
                         Third : t -> c
```

The dual of a list is a colist. Colists are lists which can be either finite or infinite. Colists are defined using the Maybe datatype as follows:

```
codata c -> CoList a = dest : c -> Maybe (a * c)
```

The data definition for infinite lists is given below. Note that an infinite list cannot be finite.

```
codata c -> InfList a = Head : c -> a
                        Tail : c -> c
```

3.2.1 Destructors

Destructors are dual to constructors. To apply a destructor, `D` to a term, `t` we use: `D t`. While constructors build data structure from pieces, destructors break down coinductive data into pieces. For example, `Triple` has three destructors: `First`, `Second` and `Third`. To obtain the types of these destructors one substitutes `t` with `Triple a b c`: thus `First` has the type `Triple a b c` $\rightarrow$ `a`, `Second` has the type `Triple a b c` $\rightarrow$ `b` and `Third` has the type `Triple a b c` $\rightarrow$ `c`. Building a codata is a bit trickier, but intuitively let `t = (a,1,‘‘First Alphabet’’)` be a triple. Then First t = a. In Figure 3.6, we use the `record` construct to create a triple.

Infinite lists have two destructors: `Head` and `Tail`. To obtain the types of the destructors, one substitutes the state variable `c` by `InfList a`: thus `Head` has the type `InfList a` $\rightarrow$ `a`, and `Tail` has the type `InfList a` $\rightarrow$ `InfList a`. The example in Figure 3.7 produces an infinite list of natural numbers : `Head 0` (`Tail` (`Head 1` (`Tail` (`Head 2` $\cdots$)))).

3.2.2 Record

The `record` operation is the counterpart of `case` for coinductive data. When using `case`, we extract the elements from a data structure and when using `record`, we construct the elements into a data structure. MPL has the following syntax to write a `record`:

$$\texttt{record}\ \texttt{d}_1 \leftarrow \texttt{t}_1$$
$$\vdots$$
$$\texttt{d}_\texttt{n} \leftarrow \texttt{t}_\texttt{n}$$

where $\texttt{t}_1 \cdots \texttt{t}_\texttt{n}$ are all terms and $\texttt{d}_1 \cdots \texttt{d}_\texttt{m}$ are destructors.

In Figure 3.6, the `record` construct is used to create a triple. Here, `makeTriple` takes three arguments and returns a triple. `First` is actually the first projection and same thing for `Second` and `Third`. The record syntax for this triple can also be thought of as `(First: a, Second: 1, Third: ``First Alphabet'')`.

```
1 makeTriple : a -> b -> c -> Triple a b c
2 makeTriple a b c =
3        record First  <- a
4               Second <- b
5               Third  <- c
```

Figure 3.6: MPL program for demonstrating the use of record

3.2.3 Unfold

The `unfold` combinator is the basic corecursion operation in sequential MPL, and is provided automatically once coinductive data is declared. The `unfold` is the only way to corecursively produce codata. It is referred to as the "anamorphism" by Meijer, et.al in [**?**, **?**].

The `unfold` has the following general form:

$$
\begin{array}{l}
\texttt{unfold} \\
\quad \texttt{f} : \texttt{T} \\
\quad \texttt{f}\ \texttt{x}_1 \cdots \texttt{x}_\texttt{p} = \\
\qquad \texttt{d}_1 \leftarrow \texttt{t}_{11} \\
\qquad\quad \vdots \\
\qquad \texttt{d}_\texttt{m} \leftarrow \texttt{t}_\texttt{m}
\end{array}
$$

where,

- $\texttt{T}$ is the type of the function $\texttt{f}$.
- $\texttt{x}_1 \cdots \texttt{x}_\texttt{p}$ are names of the parmeters of the function $\texttt{f}$.
- $\texttt{d}_1 \cdots \texttt{d}_\texttt{m}$ are destructors.
- $\texttt{t}_1 \cdots \texttt{t}_\texttt{m}$ are terms.

such that,

- $\texttt{f}$ is in the scope of $\texttt{t}_1 \cdots \texttt{t}_\texttt{m}$.
- The type of the function $\texttt{f}$ and the type of the terms $\texttt{t}_1 \cdots \texttt{t}_\texttt{m}$ are same.
- This `unfold` operation delivers a function $\texttt{f}$ with type $\texttt{T}$.

In Figure 3.7, a function is defined by using `unfold`, which generates an infinite list of natural numbers. The coinductive values for an infinte list of natural numbers are constructed. These are processed by applying destructors (`Head` and `Tail`).

3.2.4 Mutual corecursion on coinductive data

One can write mutually corecursive unfolds on non-mutually recursive coinductive data in MPL. For mutual corecursion, MPL allows many unfold phrases under the `unfold` keyword as for the inductive `fold`.

```
1 codata c -> InfList a = Head : c -> a
2                         Tail : c -> c
3 --infinite natural number list
4 unfold
5      infnats : nat -> Inflist (Nat)
6      infnats n =
7          Head <- n
8          Tail <- infnats (Succ n)
```

Figure 3.7: MPL function for giving an infinite list of natural numbers

One can also have mutual coinductive data in MPL. For mutual codata, the `unfold` must be mutually corecursive. In MPL, the `codata` keyword sets an environment in which mutually recursive codata can be declared, separated by the "and" keyword and allows sharing state variables. The general form of mutually recursive coinductive data is:

$$\begin{aligned}
&\texttt{codata}\ \mathtt{z_1} \to \mathtt{D_1}\ \widetilde{\mathtt{P}} = \mathtt{d_{11}} : \mathtt{z_1} \to \mathtt{F_{11}}\ \widetilde{\mathtt{P_{11}}}\ \mathtt{z_1} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{d_{1m}} : \mathtt{z_1} \to \mathtt{F_{1m}}\ \widetilde{\mathtt{P_{1m}}}\ \mathtt{z_1} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\text{and}\quad &\mathtt{z_n} \to \mathtt{D_n}\ \widetilde{\mathtt{P}} = \mathtt{d_{n1}} : \mathtt{z_n} \to \mathtt{F_{n1}}\ \widetilde{\mathtt{P_{n1}}}\ \mathtt{z_n} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{d_{nm}} : \mathtt{z_n} \to \mathtt{F_{nm}}\ \widetilde{\mathtt{P_{nm}}}\ \mathtt{z_n}
\end{aligned}$$

An example of mutual coinductive data for a game can be defined as follows:

```
codata c -> PGame a b  = Pmove : c ->  List (a, d)
and   d -> OGame a b  = Omove : d -> List (b, c)
```

where the codata `PGame` and `OGame` are defined in terms of each other. A typical use of this codata, is to implement a combinatoric game, such as Nim. In this game the player and

opponent are allowed to remove at least one element from one of a number of stacks. The game is lost by the player who cannot move.

3.3 Functions, function application and where

Functions are named terms in the sequential world. A function is given a name, supplied with a list of named parameters, then supplied with the term that constitutes the body of the function. The definition of a function takes the form:

$$\mathtt{f} : \mathtt{a_1} \to \cdots \to \mathtt{a_n} \to \mathtt{b}$$

$$\mathtt{f}\ \mathtt{x_1} \cdots \mathtt{x_n} = \mathtt{t}$$

where

- `f` is the name of the function.
- $\mathtt{a_1} \to \cdots \to \mathtt{a_n} \to \mathtt{b}$ is the type of the function `f`.
- `t` is a term.
- $\mathtt{x_1} : \mathtt{a_1} \cdots \mathtt{x_n} : \mathtt{a_n}$ are the parameter names which are in the scope of $\mathtt{t} : \mathtt{b}$.

In Figure 3.8, the `equal` function is defined, which checks whether two natural numbers are equal or not, in line 8-15.

To apply a function, `f`, to its arguments, $\mathtt{t_1}, \cdots, \mathtt{t_n}$, we write:

$$\mathtt{f}\ \mathtt{t_1} \cdots \mathtt{t_n}$$

In Figure 3.8, the use of function call can be seen: in line 9, where `monus` and `add` functions are called. We assume `add` function has been defined elsewhere which adds two natural numbers. The `monus` function is defined in line 2-6, using the `fold` construct.

```
1  -- computes truncated minus i.e. no negative
2  fold
3    monus : Nat -> Nat -> Nat
4    monus x y by x =
5         Zero -> y
6         Succ n -> pred (monus n y)
7  -- checks whether two numbers are equal or not
8  equal : Nat -> Nat -> Bool
9  equal x y = isZero (add (monus x y) (monus y x))
10            where
11                 isZero : Nat -> bool
12                 isZero x =
13                      case x of
14                           Zero -> True
15                           Succ n -> False
```

Figure 3.8: MPL program that checks the equality of two natural numbers

The `where` clause allows local definitions of functions in MPL like other programming languages. The `where` specifies the substitution of some terms for identifiers in a term. This corresponds to the cut rule given in message passing logic [**?**]. To substitute terms whose names are `f` and `g` from a term `t`, one uses the following syntax:

$$\texttt{t where}$$

$$\texttt{f} = \texttt{t}_\texttt{f}$$

$$\texttt{g} = \texttt{t}_\texttt{g}$$

where $\texttt{t}, \texttt{t}_\texttt{f}$ and $\texttt{t}_\texttt{g}$ are terms, `f` and `g` are names, `f` and `g` are in the scope of `t`. Note that f and g may take some parameters, for example:

$$\texttt{t where}$$

$$\texttt{f x y} = \texttt{t}_\texttt{f}$$

where `t` and $\texttt{t}_\texttt{f}$ are terms, `f` is the name of the function, `x` and `y` are names of the arguments.

Figure 3.8 highlights the use of `where` clause in the body of `equal` (line 25-32) where `equal` is defined in terms of `isZero` function which is defined under the `where` clause.

3.4 Tuples

MPL has products whose terms are tuples. MPL has the following syntax to form a tuple of terms `t1`, $\cdots$, `tn`:

$$(\mathtt{t}_1, \mathtt{t}_2, \cdots, \mathtt{t}_n)$$

If $\mathtt{T}_1$, $\mathtt{T}_2$, $\mathtt{T}_3, \cdots$, $\mathtt{T}_n$ are the types of terms $\mathtt{t}_1$, $\mathtt{t}_2$, $\mathtt{t}_3$, $\cdots$, $\mathtt{t}_n$ respectively then the type of the tuple is $(\mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3, \cdots, \mathtt{T}_n)$.

MPL facilitates the use of pattern matching a product which can be written in MPL by the following syntax:

$$\mathtt{case\ t\ of}$$
$$(\mathtt{t}_1, \mathtt{t}_2, \cdots, \mathtt{t}_n) \rightarrow \mathtt{t}_t$$

Consider the example given in figure 6.4.6 . A `swap` function is defined which takes a pair of elements and returns the swapped pair of those elements.

```
1 swap : (a,b) -> (b,a)
2 swap t =
3    case t of
4         (x,y) -> (y,x)
```

Figure 3.9: MPL program showing the use of case and pair

The `unit` is the nullary case of tuple. The unit is denoted:

$$(\ )$$

Chapter 4

Introduction to Concurrent world

The concurrent part of MPL consists of basic concurrent term constructions, protocol and coprotocol definitions. Protocols and coprotocols definitions provide user defined intial and final datatypes in the concurrent world. They arrive with their basic methods of building processes: structors (constructors and destructors), match and drive.

4.1 Basic Concurrent Terms

In this section, we describe how to build processes in MPL.

4.1.1 Defining a process

The definition of a process takes the following general form:

$$\mathtt{P} :: (\mathtt{a_1}, \cdots, \mathtt{a_n})\ \mathtt{A_1} : \mathtt{p_1}, \cdots, \mathtt{A_n} : \mathtt{p_n} \Rightarrow \mathtt{B_1} : \mathtt{q_1}, \cdots, \mathtt{B_m} : \mathtt{q_m}$$

$$\mathtt{P} = \mathtt{t_p}$$

where,

- $\mathtt{P}$ is the name of the process.
- :: seperates the name from the type of the process i.e, it can be read as P has type $(\mathtt{a_1}, \cdots, \mathtt{a_n})\ \mathtt{A_1} : \mathtt{p_1}, \cdots, \mathtt{A_n} : \mathtt{p_n} \Rightarrow \mathtt{B_1} : \mathtt{q_1}, \cdots, \mathtt{B_m} : \mathtt{q_m}$
- $\mathtt{a_1} \cdots \mathtt{a_k}$ are the types of the sequential inputs to the process.
- $\mathtt{A_1} \cdots \mathtt{A_n}$ are the names of the input channels and $\mathtt{B_1} \cdots \mathtt{B_m}$ are the names of the output channels.

- $\Rightarrow$ seperates the input and output channels.
- $\mathtt{p_1} \cdots \mathtt{p_n}$ and $\mathtt{q_1} \cdots \mathtt{q_m}$ are the names of the protocols assigned to the channels $\mathtt{A_1} \cdots \mathtt{A_n}$ and $\mathtt{B_1} \cdots \mathtt{B_m}$ respectively. The symbol, :, is used as an assignment operater which binds the type or protocol of a channel to a channel name.
- $\mathtt{t_p}$ is the process term that defines the process, `P`.
- The channel names $\mathtt{A_1} \cdots \mathtt{A_n}$ and $\mathtt{B_1} \cdots \mathtt{B_m}$ are bound in $\mathtt{t_p}$.

An MPL process, `Successor` is defined in Figure 4.1 which takes a natural number on its input channel, `inCh` and gives back the successor of that natural number on its output channel, `outCh` and then ends the communication with the `end` command. The type of the process ensures that the process peforms these general input/output actions, although the particular number which is output depends on the process. Notice, there is no sequential argument to this process which is indicated by `()` in line 1.

```
1 Successor :: () put Nat top => put Nat top
2 Successor inCh => outCh =
3    get x on inCh.
4    put (Succ x) on outCh
5    close inCh -> end outCh
```

Figure 4.1: Defining a process in MPL

4.1.2 Getting a message on a channel

The following command is used to recieve a message on a channel:

get x on X . P

where `x` is a sequential variable which is the message, X is the channel name on which the process recieves the message x, `P` is the process executed after the message is recieved. The process P can use `x`, that is, `x` is bound in `P`.

In line 3 of Figure 4.1, there is an example of a `get` operation. The message `x` is recieved on channel `inCh` and then is used by the process defined in lines 4 and 5.

When one gets a message, `x`, on a channel which has an output polarity, it must have a protocol `get A Q` where A is the sequential type of the message which is being passed and Q is the protocol of the channel after the message is recieved. On the other hand, if the channel has an input polarity, then that channel must have protocol `put A Q`.

4.1.3 Putting a message on a channel

The following general form is used to put a message on a channel:

put t on A; P

where `t` is a sequential term, the message to be passed, `A` is a channel name which is the carrier of `t`, and `P` is the process that is executed after the message has been sent.

In line 4 of Figure 4.1, there is an example of a use of `put`. From that example, we have:

put (Succ x) on outCh; close inCh → end outCh

this process outputs `Succ x` on channel `outCh` and then continues with the rest of the process which closes `inCh` and ends `outCh`, thus terminating the process.

When one outputs a message, `t` on a channel which has an output polarity, it must have a protocol `put A Q` where A is the sequential type of the message which is being passed and Q is the protocol of the channel after the message is sent. On the other hand, if the channel has an input polarity, it must have protocol `get A Q`.

4.1.4 Split and fork

MPL cannot pass channel names as messages: this, for example, is fundamental to the π-calculus [**?**]. However, MPL has other mechanisms which allow one to simulate the effect of passing channel names (see more in Example 5.3). Two important mechanisms in this regard are `split` and `fork` which allow one to bundle multiple channels into one channel and to unbundle them.

To split a channel which has an input polarity it must have protocol P $\otimes$ Q, the "tensor" of P and Q. To split a channel which has an output polarity, on the other hand, it must have protocol P $\oplus$ Q, the "cotensor" of P and Q. To fork a channel which has an output polarity it must have protocol P $\otimes$ Q, the "tensor" of P and Q. On the other hand, to fork a channel which has an input polarity, it must have protocol P $\oplus$ Q, the "cotensor" of P and Q.

MPL has the following syntax for splitting a channel:

split A into (A1, ... , An) $\rightarrow$ p

where $\mathtt{p}$ is a process, $\mathtt{A}$ is the channel to be split and $\mathtt{A_1} \cdots \mathtt{A_n}$ are channels into which channel $\mathtt{A}$ is split. Note, $\mathtt{p}$ is in the scope of $\mathtt{A_1} \cdots \mathtt{A_n}$, so may use the introduced channel names $\mathtt{A_1} \cdots \mathtt{A_n}$. However $\mathtt{p}$ is now no longer in the scope of $\mathtt{A}$, so may not use A.

MPL has the following syntax for forking channel:

$$
\begin{aligned}
&\texttt{fork A as} \\
&\quad \mathtt{A_1}\ \texttt{with}\ \mathtt{B_1} \rightarrow \mathtt{p_1} \\
&\quad \cdots \\
&\quad \mathtt{A_n}\ \texttt{with}\ \mathtt{B_n} \rightarrow \mathtt{p_n}
\end{aligned}
$$

where $\mathtt{p_1} \cdots \mathtt{p_n}$ are processes, $\mathtt{A}$ is the channel to be forked and $\mathtt{A_1} \cdots \mathtt{A_n}$ are channels into which channel $\mathtt{A}$ is forked. $\mathtt{A_i}$ is in the scope of process $\mathtt{p_i}$, but $\mathtt{A}$ is not in the context of any of $\mathtt{p_1} \cdots \mathtt{p_n}$. $\mathtt{B_1} \cdots \mathtt{B_n}$ are optional channel names that can specify the opposite polarity of $\mathtt{A_1} \cdots \mathtt{A_n}$ for $\mathtt{p_1} \cdots \mathtt{p_n}$ respectively.

A simple process, `IdPar` is defined in Figure 4.2 which is the identity for the cotensor protocol, $\oplus$. This process has one input channel, `a` and one output channel, `b`. The defnition of the process is given using split and fork: in line 3, the output channel `b` is split into two channels `b1` and `b2`. Then, the input channel is forked, in line 3 to 5, which allows the two resulting processes to run in parallel.

```
1 IdPar :: () ($A (+) $B) => ($A (+) $B)
2 IdPar a => b =
3       split b into (b1,b2) -> fork a as
4             a'  with b1 -> a' == b1
5             a'' with b2 -> a'' == b2
```

Figure 4.2: MPL process demonstrating the use of split and fork operation

A simple process, `PAppend` is defined in Figure 4.3 which "appends" at the concurrent level two lists of channels of protocol `PList $A $B`. The user defined protocol, `PList $A $B`, has two constructors: `#PCons` and `#PNil`, which allow one to bundle an arbitary number of channels together. The `#PCons` constructor allows one to add a channel to a bundle and the `#PNil` constructor defines the action when a process gets an empty list with channel identification discussed later. The `PAppend` has one channel of the input polarity, `ch1`, and one channel of output polarity, `ch2`, where input channel is split into two channels followed by a forking operation on the output channel.

4.1.5 Closing and ending channels

One is allowed to close an input polarity channel when its protocol is `top` or when it is an output polarity channel with protocol `bottom`. Provided all other channels have been closed one can end the remaining channel provided either it has protocol `top` and an output polarity or protocol `bottom` and an input polarity. This ofcourse terminates the process.

```
1  protocol PList $A $B => $C = #PCons :: $A, $C => $C
2                              #PNil  :: $B => $C
3
4  drive
5    PAppend :: () (PList () $A (PList () $A $B)) => (PList () $A $B)
6    PAppend ch1 => ch2 by ch1 =
7            #PCons : #PCons on ch2
8                     split ch1 into (c11,c12) ->
9                            fork ch2 as
10                               c21  with c11 -> c11 == c21
11                               c22  with c12 -> call Append c12 => c22
12           #PNil : ch1 == ch2
```

Figure 4.3: MPL process demonstrating the use of split and fork operation

The syntax for closing a channel in MPL is as follows:

$$\text{close A} \rightarrow \text{P}$$

where `A` is the channel name and `P` is a process. The `P` is no longer in the scope of `A`. Note, closing a channel is equivalent to:

$$\text{split A into } () \rightarrow \text{P}$$

The `end` must be the last statement of the process as everything else must be closed and there can be no process to continue. The syntax used to end a channel is as follows:

$$\text{end A}$$

where `A` is the channel name.

The use of `close` and `end` can be seen in line 5 of Figure 4.1

$$\text{close inCh} \rightarrow \text{end outCh}$$

where the channel `inCh` is closed followed by the process

end outCh

where the channel `outCh` has ended its communication after all other channel associated with the communication are closed like channel `inCh`.

4.1.6 Channel identification and negation

One can identify two channels `x` and `y` provided they have the same type and opposite polarity. Channel identification has the following form:

x == y

An example of the use of channel identification is in line 10 and 12 of Figure 4.3.

A useful feature allows the identification of two channels of the *same* polarity. In this case the type of one channel must be the "negation" of the other.

x == neg (y) or, neg (x) == y

The use of negating a channel can be seen in Example 5.3. Note: $(\alpha == \beta)$ means the same as $(\beta == \alpha)$.

4.1.7 Calling defined processes

The general form of calling a process in MPL is as follows:

call P x ⇒ y

where `P` is a process, `x` is a list of input channels and `y` is a list of output channels. A process must be called with inputs and outputs of the right type. In Figure 4.3, there is an example of a process call, the `call Append c12 ⇒ c22` in line 11.

4.1.8 Defining local functions and processes using `where`

The `where` construct in the concurrent world allows local definitions (function or process definitions) under a process definition. The `where` construct specifies the substitution of some terms for variables in the context of a process. To substitute the named terms `f` and `g` into a process `p`, one uses the following syntax:

$$\begin{aligned} &\texttt{p where} \\ &\quad \texttt{f} = \texttt{t}_1 \\ &\quad \texttt{g} = \texttt{t}_2 \end{aligned}$$

where $\texttt{t}_1$ and $\texttt{t}_2$ are terms, `p` is a process, `f` and `g` can be the names of functions or processes, and `f` and `g` are in the scope of `p`.

In Figure 4.4, the process `GetHead` takes a list on its input channel `ch`, gives back the head of that list, by using the `head` function on the same channel, and then ends the communication. In line 5, we can see that to extract the `head` of the list `x`, the process `head` is defined locally under a where clause.

```
1  GetHead :: () (put (List a) (get (Maybe a) bottom)) =>
2  GetHead ch => =
3     get x on ch. put (head x) on ch; end ch
4        where
5            head : List a -> Maybe a
6            head xs = case xs of
7                Nil       -> Nothing
8                Cons x xs -> Just x
```

Figure 4.4: MPL process demonstrating the use of where

4.1.9 Plugging processes together

The `plug` construct is used to connect two processes on a communication channel. For plugging one process to another process, the language has the following syntax:

$$\begin{aligned}&\texttt{plug on } (\texttt{A}_\texttt{k} == \texttt{A}_\texttt{p})\\ &\quad \texttt{P}_1\ \texttt{A}_{11} \cdots \texttt{A}_{1n} \Rightarrow \texttt{A}_{21} \cdots \texttt{A}_{2n}\\ &\quad \texttt{to}\\ &\quad \texttt{P}_2\ \texttt{A}_{31} \cdots \texttt{A}_{3n} \Rightarrow \texttt{A}_{41} \cdots \texttt{A}_{4n}\end{aligned}$$

where

- $\texttt{P}_1$ and `P2` are process names.
- $\texttt{A}_\texttt{k}$ and $\texttt{A}_{\texttt{i}1} \cdots \texttt{A}_{\texttt{i}n}$ (for $\texttt{i} \in \{1, 2, 3, 4\}$) are the channel names.
- $\texttt{A}_\texttt{k}$ is one of the $\texttt{A}_{21} \cdots \texttt{A}_{2n}$.
- $\texttt{A}_\texttt{p}$ is one of the $\texttt{A}_{41} \cdots \texttt{A}_{4n}$.

such that

- $\texttt{A}_\texttt{k}$ and $\texttt{A}_\texttt{p}$ carry same protocol but different polarity.

Recall the example given in Figure 2.2 where the following composite process, `TwoAverage` is created by plugging two opposite polarity channels of two different processes (`Average` and `TwoWayTalk`); the channels have to have the same type.

4.2 User defined protocols

As in the sequential world, MPL separates datatypes at the concurrent level into two classes: protocols and coprotocols. However, the concurrent world is much more symmetric thus, for example, folds and unfolds behave in the same way. Thus in the concurrent world, the

```
1 TwoAverage :: () Talk ((a * b) c) => Talk (a b), Talk (a b)
2 TwoAverage w => y, z  =
3           plug on x
4                Average w => x
5                to
6                TwoWayTalk x => y, z
```

Figure 4.5: MPL process demonstrating the plugging

constructions are same and we talk of "driving" (instead of folding or unfolding), "matching" (instead of case or record).

4.2.1 Protocol declarations

The general form of a protocol declaration is:

$$\texttt{protocol D } (\widetilde{\texttt{P}})\ \widetilde{\texttt{\$Y}} \Rightarrow \texttt{\$Z} = \#\texttt{C}_1 :: \texttt{F}_1\ \widetilde{\texttt{P}}_1\ \widetilde{\texttt{\$Y}}\ \texttt{\$Z} \Rightarrow \texttt{\$Z}$$
$$\vdots$$
$$\#\texttt{C}_\texttt{n} :: \texttt{F}_\texttt{n}\ \widetilde{\texttt{P}}_\texttt{n}\ \widetilde{\texttt{\$Y}}\ \texttt{\$Z} \Rightarrow \texttt{\$Z}$$

where,

1. D is the name of the protocol.
2. $\widetilde{\texttt{P}}$ is a sequence of sequential type parameters such as $\texttt{p}_1 \cdots \texttt{p}_\texttt{m}$ and $\widetilde{\texttt{P}}_\texttt{i}$ is a sequence of sequential type parameters for $\texttt{C}_\texttt{i}$.
3. $\widetilde{\texttt{\$Y}}$ is a sequence of concurrent type parameters such as $\texttt{\$Y}_1 \cdots \texttt{\$Y}_\texttt{k}$.
4. $\#\texttt{C}_\texttt{i}$, where i ranges from 1 to n, are the names of the constructors of the protocol.
5. \$Z is the state variable.

6. $\texttt{F}_\texttt{i}\ \widetilde{\texttt{P}}_\texttt{i}\ \widetilde{\texttt{\$Y}}\ \texttt{\$Z}$ is a type in terms of $\widetilde{\texttt{P}}_\texttt{i}$, $\widetilde{\texttt{\$Y}}$ and $\texttt{\$Z}$.

The example given in Figure 4.6 defines the `User` protocol which either receives a String, then receives a natural number and then outputs a boolean value on the input channel, or ensures the propoer ending of the communication. In the figure the `Use` protocol is defined which uses the protocol, `User`, on its input channel, `x`.

```
1  protocol User => $C = #Do :: put String (put Nat (get Bool $C)) => $C
2                        #Stop :: bottom => $C
3  drive
4    Use :: () User () =>
5    Use x => by x =
6         #Do : get s on x.
7               get n on x.
8               put True on x
9               call Use x =>
10        #Stop : end x
```

Figure 4.6: MPL program defining protocol and demonstrating its use

4.2.2 Coprotocol declarations

The general form of a coprotocol declaration is:

$$\begin{aligned}\texttt{coprotocol}\ \texttt{\$Z} \Rightarrow \texttt{CD}\ (\widetilde{\texttt{P}})\ \widetilde{\texttt{\$Y}} = \#\texttt{D}_1 :: \texttt{\$Z} \Rightarrow \texttt{F}_1\ \widetilde{\texttt{P}}_1\ \widetilde{\texttt{\$Y}}\ \texttt{\$Z} \\ \vdots \\ \#\texttt{D}_\texttt{n} :: \texttt{\$Z} \Rightarrow \texttt{F}_\texttt{n}\ \widetilde{\texttt{P}}_\texttt{n}\ \widetilde{\texttt{\$Y}}\ \texttt{\$Z}\end{aligned}$$

where,

1. `CD` is the name of the coprotocol.

2. $\widetilde{\texttt{P}}$ is a sequence of sequential type parameters such as $\texttt{p}_1 \cdots \texttt{p}_\texttt{m}$ and $\widetilde{\texttt{P}}_\texttt{i}$ is a sequence of sequential type parameters for $\texttt{C}_\texttt{i}$.

3. $\widetilde{\$\mathtt{Y}}$ is a sequence of concurrent type parameters such as $\$\mathtt{Y}_1 \cdots \$\mathtt{Y}_k$.

4. $\#\mathtt{D}_i$, where `i` ranges from `1` to `n`, are the names of the destructors of the coprotocol.

5. `$Z` is the state variable.

6. $\mathtt{F}_i\ \widetilde{\mathtt{P}}_i\ \widetilde{\$\mathtt{Y}}$ `$Z` is a type in terms of $\widetilde{\mathtt{P}}_i$, $\widetilde{\$\mathtt{Y}}$ and `$Z`.

An example of coprotocol is `CoUser`, the dual of the protocol `User` described earlier. The only difference is that we can recurse on this coprotocol on an output channel of a process while `User` can only be used to recurse on an input channel. This coprotocol has two destructors namely, `#Od` and `#Pots`. The `#Od` destructor of the coprotocol will receive a String type message at first, then receive a natural number and then output a boolean value on the same output channel, `y` which is shown in process, `CoUse`. The dual action will happen in the input channel in the same sequence if the coprotocol is applied on an input channel. The other destructor of the `CoUser` coprotocol `#Pots` ensures the proper closing and ending of the channels involved in communication.

```
1 coprotocol $C => CoUser = #Od :: $C => get String (get Nat (put Bool $C))
2                           #Pots :: $C => top
3 drive
4    CoUse :: ()  => CoUser ()
5    CoUse  => y by y =
6         #Od : get m on y.
7               get p on y.
8               put True on y
9               call CoUse => y
10        #Pots : end y
```

Figure 4.7: MPL program declaring a coprotocol and a process using it

4.2.3 Structors

In MPL, concurrent constructors and destructors behave in similar way and are called "structors". They are written starting with a hash symbol, `#`.

A protocol has a set of constructors. For example, the `User` protocol in Figure 4.6 has two constructors: `#Do` and `#Stop`. To obtain the types of these constructors, one substitutes the state variable `$C` by `User`: thus `#Do` has the type

$$\texttt{put String (put Nat (get Bool User))} \Rightarrow \texttt{User},$$

and `#Stop` has the type

$$\texttt{top} \Rightarrow \texttt{User}.$$

Destructors are the dual of constructors. The `CoUser` coprotocol in Figure 4.7 has two destructors: `#Od` and `#Pots`. To obtain the types of these destructors, one substitutes the state variable `$C` by `CoUser`: thus `#Od` has the type

$$\texttt{CoUser} \Rightarrow \texttt{put String (put Nat (get Bool CoUser))},$$

and `#Stop` has the type

$$\texttt{CoUser} \Rightarrow \texttt{top}.$$

By applying constructors, actions can be unwrapped on a channel where the protocol is assigned. For applying a constructor on a channel, MPL has the following general form:

```
#C on A; p
```

where `A` is a channel, #`C` is protocol constructor and `p` is a process.

MPL has the similar general form for applying a destructor on a channel:

```
#D on A; p
```

4.2.4 Matching structors

Match allows pattern matching on protocols. It unwraps the structors of a protocol to a channel. The general form to write the `match` is:

$$\texttt{match A as}$$
$$\#\texttt{C}_1 : \texttt{p}_1$$
$$\vdots$$
$$\#\texttt{C}_n : \texttt{p}_n$$

where

- `A` is a channel name
- $\#\texttt{C}_1 \cdots \#\texttt{C}_n$ are protocol constructors
- $\texttt{p}_1 \cdots \texttt{p}_n$ are processes

The example in Figure 4.8 demonstrates the use of `match` where `Mirror` receives and reflects a message on its input channel. In this example, a protocol named `Reflector` is defined and followed by the `Mirror` process definiton. `Mirror` has one input channel `user` carrying the `Refelector` protocol. Matching on a channel by `match` reveals its constructor `#Reflect` and unwraps it on `user`.

```
1 protocol Reflector (a) => $C =
2          #Reflect :: put a (get a BOTTOM) => $C
3 Mirror :: () Reflector(a) =>
4 Mirror user => =
5   match user as
6        #Reflect: get n on user . put n on user ; end user
```

Figure 4.8: MPL process demonstrating the use of match operation

4.2.5 Recursive processes: using the `drive` command

In the sequential world, MPL has fold for inductive datatypes and unfold for coinductive datatypes. But in the process world, both folds and unfolds use one operation, `drive`, because of the symmetry in the concurrent world; every protocol gives a coprotocol by choosing the channel (and vice-versa). MPL has the following syntax of `drive`:

```
drive
  P :: P_T
  P by A =
    #C_1 : P_1
      ⋮
    #C_m : P_m
```

where

- `P` is a process,
- $\texttt{P}_\texttt{T}$ is the type of a process,
- $\texttt{P}_1 \cdots \texttt{P}_\texttt{m}$ are processes,
- `A` is a channel name by which the process is driven and it is one of the input or output channel associated with the process,
- $\#\texttt{C}_1 \cdots \#\texttt{C}_\texttt{m}$ are protocol constructors.

such that,

- `P` is in the scope of $\texttt{P}_1 \cdots \texttt{P}_\texttt{m}$.
- This drive operation delivers a term `P`.

The example in Figure 4.2.5, defines the `Counter` process using `drive`, which can either add any number with the previous state or can ouput the present state. The `Counter` process is driven by channel `pinger` where the `Ping` protocol is applied on channel `pinger`. The process is called recursively in line 7.

```
1 protocol Ping => $C = #Ping :: put Nat $C => $C
2                       #Pong :: get Nat bottom => $C
3
4 drive
5   Counter :: (Nat) Ping() =>
6   Counter (n) pinger =>  by pinger =
7         #Ping: get i on pinger . call Counter (add n i) pinger =>
8         #Pong: put n on pinger; end pinger
```

4.3 Mutual recursion on protocols

One can write a mutually recursive drive on a non-mutually recursive protocol in MPL. For mutual recursion, MPL allows many drive phrases under the keyword, `drive`. An example is given in Figure 4.9: this example counts half of the numbers that it encounters. The multiple `drive` phrases are used to define `Tally` and `NoTally` in a mutually recursive way. These processes are driven by input channel `feed` (as `feed` is the only channel).

One can also have mutually recursive protocols in MPL. For mutually recursive protocols, the `drive` must be mutually recursive. In MPL, the `protocol` keyword sets an environment in which mutually recursive data can be declared, separated by the "and" keyword which allows sharing state variables.

The general form of a declaration of mutually recursive protocol is:

$$\texttt{protocol}\ \mathtt{D}_1\ \widetilde{\mathtt{P}}\ \widetilde{\$\mathtt{Y}} \Rightarrow \$\mathtt{Z}_1 = \#\mathtt{C}_{11} :: \mathtt{F}_{11}\ \widetilde{\mathtt{P}_{11}}\ \widetilde{\$\mathtt{Y}}\ \mathtt{Z}_1 \Rightarrow \$\mathtt{Z}_1$$
$$\vdots$$

```
1  protocol Counter => $C = #More   :: $C => $C
2                          #Nomore :: get Nat bottom => $C
3 {-
4   This is the process which counts half of the numbers it encounters
5 -}
6    drive
7       Tally (n) :: () Counter () =>
8       Tally (n) feed => by feed =
9             #More:   call NoTally (Succ n) feed =>
10            #Nomore: put n on feed; end feed
11      NoTally (n) :: () Counter () =>
12      NoTally (n) feed => by feed =
13           #More:   call Tally (n) feed =>
14           #Nomore: put n on feed ; end feed
```

Figure 4.9: MPL program demostrating mutual recursion

$$\#\mathtt{C_{1m}} :: \mathtt{F_{1m}}\ \widetilde{\mathtt{P_{1m}}}\ \widetilde{\$\mathtt{Y}}\ \mathtt{Z_1} \Rightarrow \$\mathtt{Z_1}$$

$$\vdots$$

$$\text{and} \quad \mathtt{D_n}\ \widetilde{\mathtt{P}}\ \widetilde{\$\mathtt{Y}} \Rightarrow \$\mathtt{Z_n} = \#\mathtt{C_{n1}} :: \mathtt{F_{n1}}\ \widetilde{\mathtt{P_{n1}}}\ \widetilde{\$\mathtt{Y}}\ \mathtt{Z_n} \Rightarrow \$\mathtt{Z_n}$$

$$\vdots$$

$$\#\mathtt{C_{nm}} :: \mathtt{F_{nm}}\ \widetilde{\mathtt{P_{nm}}}\ \widetilde{\$\mathtt{Y}}\ \mathtt{Z_n} \Rightarrow \$\mathtt{Z_n}$$

An example showing the use of mutual recursion on mutual protocol is given in Figure 4.10 which implements the copycat strategy. The mutual protocol for the copycat strategy is defined in lines 1-4. The processes `Talk` and `Listen` are defined using mutually recursive drive. The constructor #`response` for the `Talk`(`a` `b`) protocol indicates that the `Talk` process will receive `a` in the input channel and then call the `Respond`(`a` `b`) protocol. On the other hand, the constructor #`listen` for the `Respond`(`a` `b`) protocol indicates that the process will receive `b` in the output channel, and then call the `Talk`(`a` `b`) protocol.

```
1   protocol Talk (a b) => $C =
2              #response :: put a $D => $C
3   and      Respond (a b) => $D  =
4              #listen :: get b $C => $D
5
6   drive
7     Talk :: () Talk (a b) => Talk (a b)
8     Talk x => y by x =
9           #response : #response on y
10                      get a on x.
11                      put a on y
12                      call Listen x => y
13    Listen :: () Respond (a b) => Respond (a b)
14    Listen x => y by x =
15          #listen : #listen on y
16                    get b on y.
17                    put b on x
18                    call Talk x => y
```

Figure 4.10: MPL program demostrating mutual recursion on mutual protocol

4.4 Mutual corecursion on coprotocols

Like, mutual protocols, in MPL, we can also build mutually corecursive coprotocols. In MPL, the `coprotocol` keyword sets an environment in which multiple coprotocols can be declared at once, seperated by the "and" keyword, and allows sharing state variables. The general form of a mutually recursive coprotocol is:

$$
\begin{aligned}
&\texttt{coprotocol}\ \$\mathtt{Z}_1 \Rightarrow \mathtt{CD}_1\ \widetilde{\mathtt{P}}\ \widetilde{\$\mathtt{Y}} = \#\mathtt{D}_{11} :: \$\mathtt{Z}_1 \Rightarrow \mathtt{F}_{11}\ \widetilde{\mathtt{P}_{11}}\ \widetilde{\$\mathtt{Y}}\ \$\mathtt{Z}_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \#\mathtt{D}_{1\mathtt{m}} :: \$\mathtt{Z}_1 \Rightarrow \mathtt{F}_{1\mathtt{m}}\ \widetilde{\mathtt{P}_{1\mathtt{m}}}\ \widetilde{\$\mathtt{Y}}\ \$\mathtt{Z}_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\texttt{and}\qquad\qquad \$\mathtt{Z}_\mathtt{n} \Rightarrow \mathtt{CD}_\mathtt{n}\ \widetilde{\mathtt{P}}\ \widetilde{\$\mathtt{Y}} = \#\mathtt{D}_{\mathtt{n}1} :: \$\mathtt{Z}_\mathtt{n} \Rightarrow \mathtt{F}_{\mathtt{n}1}\ \widetilde{\mathtt{P}_{\mathtt{n}1}}\ \widetilde{\$\mathtt{Y}}\ \$\mathtt{Z}_\mathtt{n} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{aligned}
$$

$$\#\mathtt{D}_{\mathtt{nm}} :: \$\mathtt{Z}_{\mathtt{n}} \Rightarrow \mathtt{F}_{\mathtt{nm}}\ \widetilde{\mathtt{P}_{\mathtt{nm}}}\ \widetilde{\$\mathtt{Y}}\ \$\mathtt{Z}_{\mathtt{n}}$$

An example showing the use of mutual corecursion on mutual coprotocol is given in Figure 4.11 which also implements the copycat strategy but in this example we recurse on the opposite polarity channel than that of Figure 4.10. The mutual coprotocol for copycat strategy is defined in line 1-4. The processes `CoTalk` and `CoListen` are defined using mutually recursive drive. The destructor #`coresponse` for the `CoTalk`(`a b`) coprotocol indicates that `CoTalk` process will receive `a` in the input channel and then call `CoRespond`(`a b`) coprotocol. On the other hand, the destructor #`colisten` for the `CoRespond`(`a b`) coprotocol indicates that the process will receive `b` in the output channel, and then call `CoTalk`(`a b`) coprotocol.

```
1   coprotocol $C => CoTalk (a b) =
2          #coresponse :: $C => put b $D
3   and  $D => CoRespond (a b) =
4          #colisten :: $D => get a $C
5
6   drive
7     CoTalk :: () CoTalk (a b) => CoTalk (a b)
8     CoTalk x => y by y =
9           #coresponse : #coresponse on x
10                      get a on x.
11                      put a on y
12                      call CoListen x => y
13    CoListen :: () CoRespond (a b) => CoRespond (a b)
14    CoListen x => y by y =
15          #colisten : #colisten on x
16                    get b on y.
17                    put b on x
18                    call CoTalk x => y
```

Figure 4.11: MPL program demostrating mutual recursion on mutual coprotocol

Chapter 5

Examples in MPL

5.1 Bank Machine

Consider what happens when someone wants to withdraw some money from a bank machine (or ATM). They normally insert their card into the bank machine, with the required personal identification number, and a money request. The bank machine sends this information to the bank. The bank then uses this information to generate a transaction identification number and to check whether the personal identification number is correct and whether the requested amount is available in the account. The bank machine then does a security check using the transaction identification number and account number. Finally, an acknowledgement from the bank machine will be sent to the bank on successful completion or failure of the transaction.

To develop an MPL program for a bank machine the first step is to understand the communication channels which are involved. These are illustrated in Figure 5.1.

The next step is to define the protocols that will be used on these channels. To define these protocols one needs to understand the information which must pass on the channels. The bank machine will get the personal identification number, PIN (we shall assume the PIN contains the account number), and a money request, REQ, on channel usr. The bank responds with a transaction identification number, TIN, account number, ACCOUNT, and whether the account has the required funds. The bank machine checks with the security system on channel, `sec`, to ensure the transaction is legitimate (i.e. the bank card has not been reported stolen etc.). These data are defined in Figure 5.2.

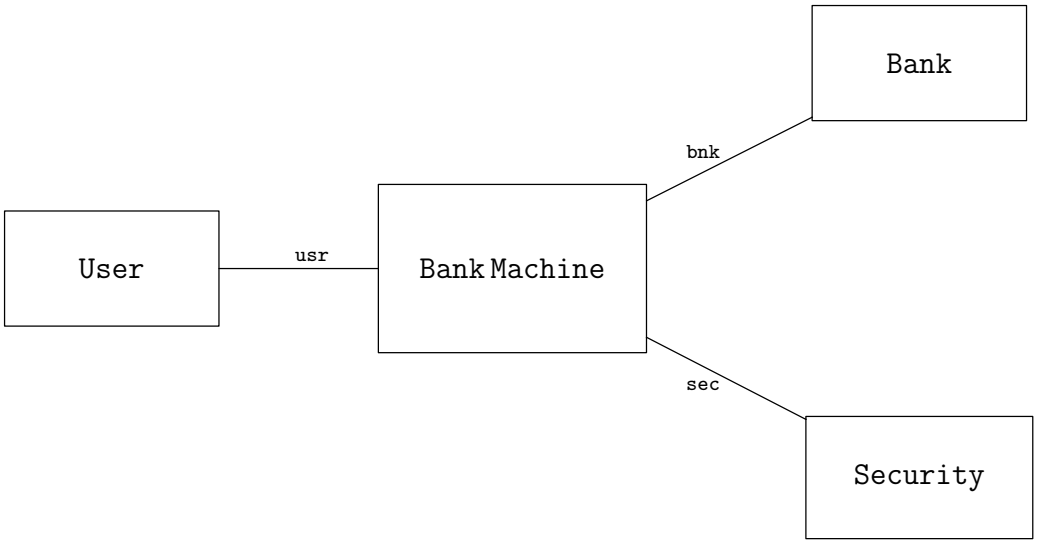


Figure 5.1: Banking Machine interaction channels

The next step is to define the protocols which will be used on each channel. The protocol on channel `bnk` is `Bank`, the protocol on channel `sec` is `Security` and the protocol on channel `usr` is `User`: these are given in Figure 5.3. Here we assume that the data `PIN, REQ, TIN` and, `ACCOUNT` are defined. Each protocol determines the sequence of possible actions on the channel to which it is assigned. Protocols allow infinite interactions so that after a transaction, the bank machine can be left ready for the next transaction.

Having defined the channels and their protocols we can write the MPL program which is given in Figure 5.4. Note, this program shows how the messages of the sequential world can control events in the concurrent world (line 13-22). The message received from Bank, `Money` or `NoMoney` and the message received from Security, `Accept` or `Deny` determines the subsequent behaviour of the process.

```
1  --  bank's response to bank machine which indicates
2  --  the availability of money
3  data MResponse -> c = Money : c
4                        NoMoney : c
5  --  bank machine's response to user which idicates
6  --  whether to hold or return the card
7  data Response -> c  = TakeCard : c
8                        GiveCard :c
9  -- security's response to bank machine which
10 -- indicates whether the transaction is legitimate or not
11 data SResponse -> c = Accept : c
12                       Deny : c
13 -- bank machine's response to bank which
14 -- indicates whether the transaction is successful or Failed
15 data Back -> c = Success : c
16                  Fail : c
```

Figure 5.2: Data, necessary to pass on channels

5.2 Diffie-Hellman key exchange

Consider an example from the field of cryptography: the Diffie-Hellman key exchange protocol. This allows Alice and Bob to establish a shared secret key in order to be able to pass encoded messages through an insecure channel. The key exchange protocol between Alice and Bob works as follows:

- Alice and Bob agree to use two numbers $p = 7$ and $g = 3$ where p is prime and g is a primitive root mod p. For our example, set $p = 7$ and $g = 3$.
- Alice choose a secret number a, computes $A = g^a \ mod \ p$ and sends A to Bob. Thus if Alice chooses $a = 4$, $A = 3^4 \ mod \ 7 = 4$.
- Bob choose a secret number b, computes $B = g^b \ mod \ p$ and sends B to Alice. Thus if Bob chooses $b = 5$, $B = 3^5 \ mod \ 7 = 5$.
- Alice computes the secret key, $s = B^a \ mod \ p$. Here, $s = (5)^4 \ mod \ 7 = 2$.

```
1  -- protocol for usr channel
2  protocol User => $C =
3   #use :: put PIN (put REQ (get REQ (get Response $C))) => $C
4  -- protocol for bnk channel
5  protocol Bank => $C =
6   #bank :: put (PIN * REQ)
7            (get TIN (get ACCOUNT (get MResponse (put Back $C)))) => $C
8  -- protocol for sec channel
9  protocol Security => $C =
10  #security :: put TIN (put ACCOUNT (get SResponse $C)) => $C
```

Figure 5.3: Protocols for channels

- Bob computes the secret key, $s = A^b \ mod \ p$. Here, $(4)^5 \ mod \ 7 = 2$.
- Alice sends an encrypted message to Bob using the secret key.

We develop an MPL program for the Diffie-Hellman key exchange protocol. The first step is to understand the communication channels which are involved. These are illustrated in figure 5.5. Alice has three channels associated with her. Alice uses two secure channels to receive respectively a secret key and the message to be encoded: these are `secretA` and `messageA` respectively. Bob also has three channels: two secure channels to receive a secret key and to send the decoded message after receiving it from Alice on the insecure channel. Alice and Bob communicate on the insecure channel, `cipher`.

The next step is to determine the protocols for these channels so that the desired actions can take place on each channel. The protocols are defined:

- `secretA` is a secure channel on which Alice can obtain a secret key. The protocol we shall use is `put key top`. This allows Alice to get the secret key and then close the communication.
- `messageA` is a secure channel on which Alice can receive a message to be sent to Bob. The protocol we shall use is `put msg top`. This allows Alice to get

the message to be encoded and then close the communication.

- `cipher` is an insecure channel shared between Alice and Bob. The protocol we shall use is `put key (get key (put msg top))`. This allows Alice to send her generated key to Bob, then Bob to send his generated key to Alice, and then, for Alice to send the encoded message to Bob and close the communication.

- `secretB` is a secure channel on which Bob can obtain a secret key. The protocol we shall use is `get key bottom`. This allows Bob to get the secret key and then close the communication.

- `messageB` is a secure channel on which Bob can send the decoded message on. The protocol we use is `put msg top`. This allows Bob to send the decoded message and then end the communication.

We assume the types `key` and `msg` have already been defined. We also assume the functions `modulus`, `power`, `encode` and `decode` have been defined elsewhere.

Having defined the channels and their types we can write the MPL program which is given in Figure 5.6. The protocols are used in the process type of Alice and Bob. Note the two numbers that Alice and Bob agreed to use are sequential arguments to their processes.

5.3 Memory passing: Mobility

This example illustrates how the topology of the communicating processes in MPL can be changed dynamically. The example we will present is similar to the mobile phone example of Chapter 8 in [**?**]. To illustrate mobility, we shall show how a "mutable" memory cell can be passed between two processes in MPL. The example used ideas of Emmanuel Beffara [**?**].

To develop this MPL program, the first step is to write the process for a memory cell. To start with we need to understand the protocol associated with the communication channel of a memory cell. A memory cell can either remember a value which is passed to it or it can provide, on demand, the value it is remembering. A memory cell, as in Figure 5.7, communicates to the outside world through a single channel: through this channel it can receive values and output (remembered) values. A memory cell should be able to repeatedly receive new values and provide the value it has remembered. In order to program these actions a protocol for the interface to a memory cell, named `Memory`, is defined, see lines 1-3 in Figure 5.8. The process `MemoryCell`, using the `Memory` protocol, on channel `ch` is in lines 4-8 of Figure 5.8.

When the protocol, `Memory` is connected on the channel, `ch`, the channel behaviour is determined by the constructor, #`rcv` or #`snd`. which is received. The memory cell process reads the value when the constructor is #`rcv` and remembers the recieved value. On the other hand, on the #`snd` constructor the process memory cell returns the remembered value along the channel, `ch`, and still remembers the same value.

Our aim is to mobilize this memory cell (see Figure 5.9) between two processes (P1 and P2) so that the memory cell can communicate alternately with one process then the other.

Each process (`P1` or `P2`) can change the value in the memory cell and can also retrieve stored values from the memory cell. These actions are determined here by the protocol, `Talker`, connected to the process in Figure 5.10. The moving of a memory cell between two processes can be done by defining a protocol `Passer`. The process `P1` has two input polarity channels, `c1` and `in2` and one output polarity channel, `c2` and is driven by `c1`. The `Passer` protocol is applied on `c1`, the `Talker` protocol is applied on `in2` and the protocol `Memory` is applied on `c2`. The `P1` process extracts a stored value from the memory cell and then puts a new value which is receieved on channel `in2` into the memory cell. `P1` then passes

the extracted value on its channel in2. After that, the memory cell is passed to the other process `P2`. The process, `P2`, has one input polarity channel, `in1` and one output polarity channel, `c` and is driven by `in1`. `P2` also extracts value from the memory cell which was stored by `P1` and passes it back along channel in2 again after storing a new value in the memory cell. Each time the memory cell is passed back and forth between processes, the protocol, `Memory`, needs to be negated. This program gives a simple example to show that the connection topology of a MPL program need not be fixed but can be quite dynamic.

```
1  drive
2    BankMachine :: () User () => Bank (), Security ()
3    BankMachine usr => bnk, sec by usr =
4       #use : get pin on usr.
5              get req on usr.
6              #bank on bnk
7              put (pin * req)  on bnk
8              get tin on bnk.
9              get account on bnk.
10             get moneyResponse on bnk.
11             case moneyResponse of
12               Money -> #security on sec
13                           put tin on sec
14                           put account on sec
15                           get srp on sec.
16                           case srp of
17                            Accept -> put i on usr
18                                      put GiveCard on usr
19                                      put Success on bnk
20                                      call BankMachine usr => bnk, sec
21                            Deny ->   put Zero on usr
22                                      put TakeCard on usr
23                                      put Fail on bnk
24                                      call BankMachine usr => bnk, sec
25               NoMoney ->  put Zero on usr
26                           put GiveCard on usr
27                           put Fail on bnk
28                           call BankMachine usr => bnk, sec
```

Figure 5.4: MPL program for bank machine

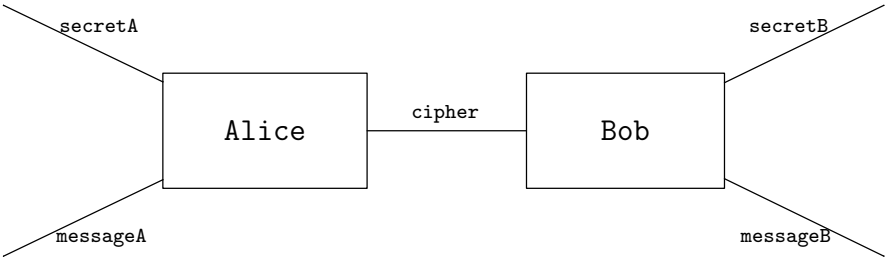


Figure 5.5: Key exchange between Bob and Alice

```
1  --DiffieHellman key exchange
2  Alice :: (key, key) put key top, put msg top
3        => put key ( get key (put msg top))
4  Alice (agreedKey1 agreedKey2) secretA, messageA => cipher  =
5          get skey on secretA.
6          put (modulus (power agreedKey2 skey) agreedKey1) on cipher
7          get bkey on cipher.
8          get msg on messageA.
9          put (encode msg (modulus (power bkey skey) agreedKey1)) on cipher
10         close secretA -> close messageA -> end cipher
11
12 Bob :: (key, key) put key (get key (put msg top))
13     => get key bottom, put msg top
14 Bob (agreedKey1 agreedKey2) cipher => secretB, messageB =
15         get skey1 on secretB.
16         get bkey1 on cipher.
17         put (modulus(power agreedKey2 skey1)agreedKey1) on cipher
18         get enmsg on cipher.
19         put decode(enmsg((modulus(power bkey1 skey1)agreedKey1))) on messageB
20         close secretB -> close cipher -> end messageB
21
22 AliceBob :: (key, key) put key top, put msg top
23          => get key bottom, put msg bottom
24 AliceBob (agreedKey1 agreedKey2) secretA, messageA => secretB, messageB =
25     plug on cipher
26         Alice (agreedKey1, agreedKey2) secretA, messageA => cipher
27         to
28         Bob (agreedKey1, agreedKey2) cipher => secretB, messageB
```

Figure 5.6: MPL program for Diffie-Hellman key exchange

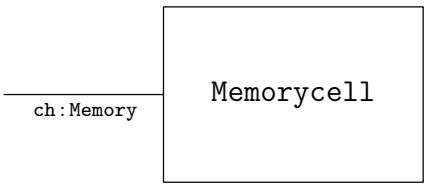


Figure 5.7: Memory Cell

```
1 protocol Memory (a) => $C =
2                #rcv :: put a $C => $C
3                #snd :: get a $C => $C
4 drive
5   Memorycell :: (a) Memory (a) =>
6   Memorycell (n) ch =>  by ch =
7             #rcv : get a on ch. call Memorycell (a) ch =>
8             #snd : put n on ch; call Memorycell (n) ch =>
```

Figure 5.8: MPL program for Memory cell

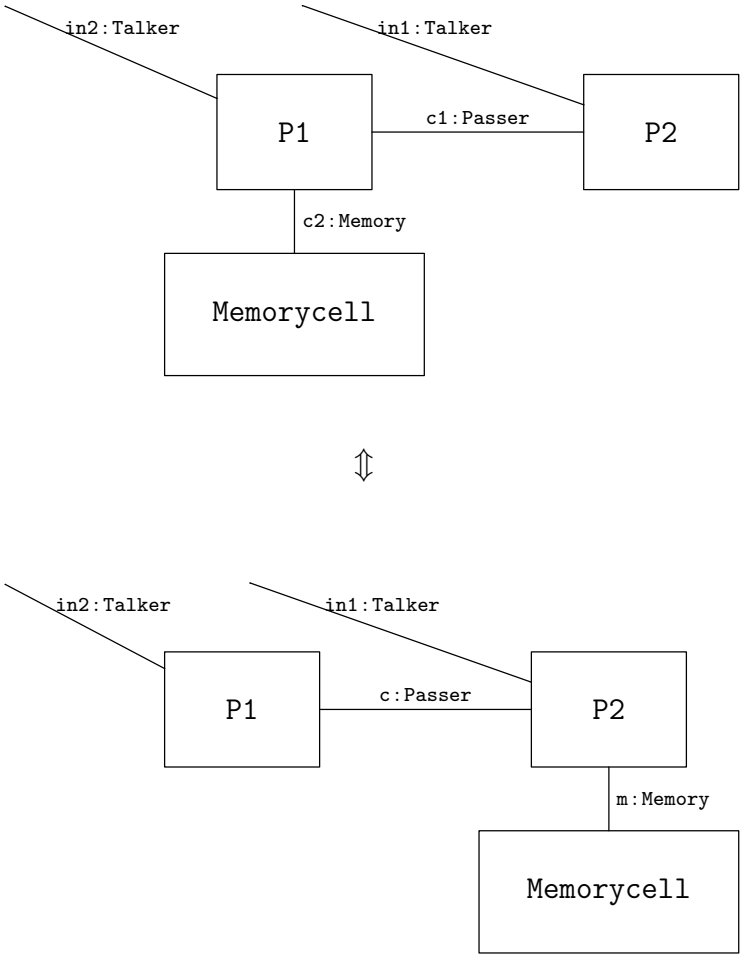


Figure 5.9: Memory cell mobility

```
1  protocol Talker (A) => $C  =
2          #talk ::  put A get A $C => $C
3
4  protocol Passer $M => $C =
5         #pass :: ($M (+) (Neg($M) (x) $C)) => $C
6
7  drive
8   P1 :: ()  Passer (Memory (A)), Talker (A) => Memory (A)
9   P1 c1 , in2  => c2 by c1 =
10    #pass: match in2 as
11            #talk : get x on in2.
12                    #snd on c2; get y on c2. #rcv on c2;
13                    put x on c2; put y on in2;
14                    fork c1 as
15                    x1 with c2 -> x1 == c2
16                    x2 with in2 -> split x2 into (nm, x3) -> plug on x1
17                                                             P1 x3, in2 => x1
18                                                             to
19                                                             (Neg (x1) == nm)
20
21 drive
22  P2 :: ()  Talker (A) => Passer (Memory (A))
23  P2  in1  => c by in1 =
24      #talk : get x on in1.
25              #pass on c;
26              split c into (m, x4) ->
27                    #snd on m; get y on m. put y on in1;
28                    #rcv on m; put x on m;
29                    fork x4 as
30                         nm with m -> (Neg (m) == nm)
31                         x5 with in1 -> call P2 in1 => x5
```

Figure 5.10: MPL program showing the mobility of a memory cell

Chapter 6

The Type System for the Sequential world

The objective of a type system is to be able to deterministically decide whether a term has a proposed type. The aim is to do this using the structure of the term alone. If this is achieved then the type system can also be used to support type inference. To achieve this it must be possible to show how a term can be uniquely decomposed into components whose types determine the typing of the whole term. This, therefore, is the objective of the type system developed below.

MPL has two layers: the sequential layer and the concurrent layer. Each layer has a type system which allows one to recognize correctly typed programs. This chapter describes the type system for the sequential layer.

The core of the sequential layer is based on a fragment of polymorphic λ-calculus [**?**] (called $\lambda 2$ in [**?**] or system-F in [**?**]). Our goal in this chapter is to introduce the fragment of the polymorphic λ-calculus used by MPL, and the extension of it used in MPL that provides initial and final data (`data` and `codata`).

The λ-calculus as conceived by Church [**?**] was developed to provide a theory of functions that could serve as the foundations of mathematics [**?**]. The untyped λ-calculus is a Turing complete language which is the basis for functional programming languages.

Church proposed a system, called the simply typed λ-calculus, which we will review in the next section. This calculus is based on the intuitive notions that a function $f : A \to B$ applied to a λ-term t of type A, written $t : A$, will return a λ-term of type B, $f\,t : B$. A formal connection between the simply typed λ-calculus and the Brouwer's propositional intuitionistic logic was described by Curry and Howard in [**?**]. In propositional intuitionistic

logic, for example, a proof of $A \Rightarrow B$ is to be regarded as corresponding to a procedure for transforming proofs of A into proofs of B that is a function $f : A \to B$ which acts on proofs.

The polymorphic typed λ-calculus is the typed λ-calculus with the additional ability to universally quantify propositional formulae [**?**]. In this chapter, we start by reviewing the simply typed λ-calculus and then describe the type system that MPL uses for its sequential types, which is based on the polymorphic λ-calculus.

6.1 The simply typed λ-calculus

The simply typed λ-calculus provides a basic example of a type system. The types are generated by:

$$T = A \mid T \to T$$

where A is a type variable (belonging to a set of type variables). For example, A, B, $A \to B$, $(A \to B) \to A$, $(A \to B) \to (C \to D)$ etc. are types.

The programs or λ-terms are generated by

$$P = x \mid \lambda x.P \mid PP$$

where x is a variable (belonging to a set of term variables), $\lambda x.P$ is an abstraction operation which binds the variable, x, in a term, P and PP is the application of a λ-term to a λ-term. Some examples of terms are $xx, xy, \lambda xy.x, \lambda xy.y$ etc. Two syntactic conventions are followed when writing λ-terms: abstractions are flattened, for example: $\lambda xy.P$ denotes $\lambda x(\lambda y.P)$ and application associates to the left, for example: xyz denotes $(xy)z$.

In order to facilitate the description of this type system the definitions of a context and a term in context are discussed first before introducing the typing rules in detail.

A **context** is a multiset of distinct typed variables, written $\Delta = x_1 : T_1, \cdots, x_n : T_n$. It is a multiset as the order does not matter and the variable names are distinct in the sense

$$\frac{\Delta, x : T, \Delta' \ \ \mathsf{context}}{\Delta, x : T, \Delta' \vdash x : T} \ \text{TermProj}$$

$$\frac{\Delta, x : T \vdash t : T'}{\Delta \vdash \lambda x.t : T \to T'} \ \lambda - \text{abst}$$

$$\frac{\Delta \vdash f : T \to T' \quad \Delta \vdash t : T}{\Delta \vdash ft : T'} \ \text{TermApplication}$$

Table 6.1: Simply typed λ-term building

that $x_i = x_j$ implies $i = j$.

A **term in context** in the simply typed λ-calculus is a sequent of the form:

$$\Delta \vdash t : T$$

where Δ is the context and t is a term with assigned type T. This is a *judgement* of the simply typed λ-calculus although it is only considered to be a *valid judgement* when it can be derived by the type system given in 6.1.

The inference rules that describe how to construct well-typed λ-terms or programs are given in 6.1. Programs in the simply typed λ-calculus are terms in context: the context collects all the symbols (variables) that the program uses and the types associated to these symbols.

The λ-`abst` rule says if $x : T$ is used in a proof of $t : T'$ so that we have:

$$\begin{array}{c} [x : T] \\ \vdots \\ t : T' \end{array}$$

then we can build a proof of $T \to T'$. This is the implication introduction rule of logic and the program that proves the implication introduction is $\lambda x.t$.

The `TermApplication` rule is the modus ponens rule or the implication elimination rule of logic. Thus, if we have a proof of $T \to T'$ and of T then we have a proof of T'.

The following example in which the `K` combinator $\lambda xy.x$ is type checked shows the use of λ-`abst` and `TermProj` rules from table 6.1 .

$$\dfrac{\dfrac{\dfrac{x : A, y : B \ \ \mathsf{context}}{x : A, y : B \vdash x : A}\ \text{TermProj}}{x : A \vdash \lambda y.x : B \to A}\ \lambda - \text{abst}}{\vdash \lambda xy.x : A \to B \to A}\ \lambda - \text{abst}$$

In the typed λ-calculus, a benefit of typing is that every typed program terminates [**?**]. For example $\lambda x.xx$ cannot be typed in the simply typed λ-calculus: when $\lambda x.xx$ is applied to itself it produces a program which does not terminate. Here is what a proof would have to look like if we were to try to infer the type of $\lambda x.xx$:

$$\dfrac{\dfrac{\dfrac{}{x : P_1 \vdash x : P_3\ \langle P_1 = P_3 \rangle}\ \text{TermProj} \quad \dfrac{}{x : P_1 \vdash x : P_4\ \langle P_1 = P_4 \rangle}\ \text{TermProj}}{x : P_1 \vdash xx : P_2\ \langle P_3 = P_4 \to P_2 \rangle}\ \text{TermApp}}{\vdash \lambda x.xx : Q\ \langle Q = P_1 \to P_2 \rangle}\ \lambda - \text{abst}$$

In this proof, an arbitrary type is assigned in each step to the terms. We get $P_1 = P_3$ and $P_1 = P_4$, by TermProj rule from the upper portion of the proof. From these two equations we get $P_3 = P_4$. In the next step, we have $P_3 = P_4 \to P_2$. Substituting P_3 by P_4 then we obtain $P_4 = P_4 \to P_2$ which has no solution because occurs check fails. Thus the term cannot be validly typed, that is, there is no valid judgement of the form $\quad \vdash \lambda x.xx : Q$, in this system.

6.2 Basic polymorphic system for sequential MPL

The polymorphic λ-calculus [**?**], [**?**] is an extension to the simply typed λ-calculus in which existential and universal quantification over type variables are allowed. This allows types

to be arguments of functions on types. In the polymorphic λ-calculus, kinds are used to describe the type quantification. For the basic polymorphic system for the sequential type system of MPL, only universal quantification and a subset of the possible kinds will be allowed. This fragment of the polymorphic λ-calculus which is the basis of sequential MPL will be reviewed.

The most basic kind, $\bullet$, denotes the simple types and the simply typed terms. A higher kind such as $\bullet \to \bullet$ will have as its "types" $F : \bullet \to \bullet$ a type which depends on a type. If F is supplied with a type T of kind $\bullet$, then it will return a type of kind $\bullet$. A higher kind term is a type function, and its kind tells us how many types it needs in order to evaluate to a type of kind $\bullet$. Kinds for sequential MPL are generated by:

$$\alpha := \bullet \mid \bullet \to \alpha$$

Typical examples of the type kinds are: $\bullet, \bullet \to \bullet, \bullet \to (\bullet \to \bullet), \cdots$. However, for example $(\bullet \to \bullet) \to (\bullet \to \bullet)$, which is allowed in polymorphic λ-calculus, is not allowed in this system.

In order to facilitate the description of the MPL type system the context of a term or program needs to be built gradually from the empty context. The rules dedicated to building contexts will be indicated with the word context in the conclusion of a rule.

The typing rules for the fragment of the polymorphic λ-calculus, used by MPL, are given in table 6.2 and the type building rules are in table 6.3. The rules given in the table 6.1 are represented in table 6.2 again with their kinds. The two new rules are TypeAbs and TypeApp. The TypeAbs rule allows one to abstract a type variable. The TypeApp rule applies a function that expects a type to a type.

Given a polymorphic term one can particularize it i.e. substitute out the type variable (TypeApp rule). In this rule, $(\Lambda X.V)@T$ can be evaluated by substituting T for X in V,

$$\frac{\Delta \vdash T : \alpha \quad new\ x}{\Delta, x : T : \alpha \quad \mathsf{context}}\ \text{TermVarIntro}$$

$$\frac{\Delta, x : T : \bullet, \Delta' \quad \mathsf{context}}{\Delta, x : T : \bullet, \Delta' \vdash x : T : \bullet}\ \text{TermProj}$$

$$\frac{\Delta, x : T : \bullet \vdash t : T' : \bullet}{\Delta \vdash \lambda x : T.t : T \to T' : \bullet}\ \lambda - \text{abst}$$

$$\frac{\Delta \vdash f : T \to T' : \bullet \quad \Delta \vdash t : T : \bullet}{\Delta \vdash ft : T' : \bullet}\ \text{TermApplication}$$

$$\frac{\Delta, X : \bullet \vdash s : S : \alpha}{\Delta \vdash \Lambda X.s : \Lambda X.S : \bullet \to \alpha}\ \text{TypeAbs}$$

$$\frac{\Delta \vdash T : \bullet \quad \Delta \vdash t : V : \bullet \to \alpha}{\Delta \vdash t@T : V@T : \alpha}\ \text{TypeApp}$$

Table 6.2: Polymorphic λ-term building

$V[T/X]$, this is a "β-reduction" of the type expression. For example, if $V = \Lambda X.X \to Y : \bullet \to \bullet$ then $V@T$ can be β-reduction to $(X \to Y)[T/X] = T \to Y$.

Notice in the TermApplication rule, the single kind ($\bullet$) ensures that programs have to have a simple type i.e. no universal quantification is present before they can be applied to any term.

Let us extend the previous example ($\lambda xy.x : A \to B \to A : \bullet$) to the polymorphic setup. In this example we show how to particularize type. Suppose we have a type, $\mathbb{Z}$. We want to specialize

$$\lambda xy.x : A \to B \to A : \bullet$$

to type, $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$. At first, we make this term depend on types. We can abstract its type

$$\frac{}{\varepsilon \ \ \mathsf{context}}\ \text{EmptyContext}$$

$$\frac{\Delta \ \ \mathsf{context} \quad new\ X}{\Delta, X : \bullet \qquad \mathsf{context}}\ \text{TypeVarIntro}$$

$$\frac{\Delta, X : \bullet, \Delta' \ \ \mathsf{context}}{\Delta, X : \bullet, \Delta' \vdash X : \bullet}\ \text{TypeProj}$$

$$\frac{\Delta \vdash T : \bullet \quad \Delta \vdash S : \bullet}{\Delta \vdash T \to S : \bullet}\ \text{Implication}$$

$$\frac{\Delta \vdash T : \bullet \quad \Delta \vdash T' : \bullet}{\Delta \vdash (T \times T') : \bullet}\ \text{TypeProduct}$$

Table 6.3: Type Building

variables: $\Lambda AB.\lambda xy.x : A \to B \to A : \bullet \to \bullet \to \bullet$. Typing this term needs several extra steps beyond the previous proof, and these steps are shown in the proof below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon \ \ \mathsf{context}}\ \text{TypeVI}}{A : \bullet \ \ \mathsf{context}}\ \mathsf{new\ A}\ \text{TypeVI}}{A : \bullet, B : \bullet \ \ \mathsf{context}}\ \mathsf{new\ B}\ \text{TypeVI}}{A : \bullet, B : \bullet \vdash A : \bullet}\ \text{TypeProj}}{A : \bullet, B : \bullet, x : A : \bullet \ \ \mathsf{context}}\ \mathsf{new\ x}\ \text{TermVI}}{A : \bullet, B : \bullet, x : A : \bullet \vdash B : \bullet}\ \text{TypeProj}}{A : \bullet, B : \bullet, x : A : \bullet, y : B : \bullet \ \ \mathsf{context}}\ \mathsf{new\ y}\ \text{TermVI}}{A : \bullet, B : \bullet, x : A : \bullet, y : B : \bullet \vdash x : A : \bullet}\ \text{TermProj}}{A : \bullet, B : \bullet, x : A : \bullet \vdash \lambda y.x : B \to A : \bullet}\ \lambda - \text{abst}}{A : \bullet, B : \bullet \vdash \lambda xy.x : A \to B \to A : \bullet}\ \lambda - \text{abst}}{A : \bullet \vdash \Lambda B.\lambda xy.x : \Lambda B.A \to B \to A : \bullet \to \bullet}\ \text{TypeAbs}}{\vdash \Lambda AB.\lambda xy.x : \Lambda AB.A \to B \to A : \bullet \to \bullet \to \bullet}\ \text{TypeAbs}$$

Now we can apply it to the type $\mathbb{Z}$ twice by using type application rule: $(((\Lambda AB.\lambda xy.x : A \to B \to A)@\mathbb{Z})\ @\mathbb{Z} : \bullet \to \bullet \to \bullet)$. It will be reduced to $\lambda xy.x : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} : \bullet$.

$$\frac{\Delta, a : A : \bullet, b : B : \bullet, \Delta' \vdash \gamma}{\Delta, (a, b) : A \times B : \bullet, \Delta' \vdash \gamma}\ \text{ProductPattern}$$

$$\frac{\Delta \vdash t_1 : T : \bullet \quad \Delta \vdash t_2 : T' : \bullet}{\Delta \vdash (t_1, t_2) : (T \times T') : \bullet}\ \text{TermProduct}$$

$$\frac{\Delta \qquad \mathsf{context}}{\Delta \vdash () : 1 : \bullet}\ \text{TermUnit}$$

$$\frac{\Delta \vdash t : T : \bullet}{\Delta, () : 1 : \bullet \vdash t : T : \bullet}\ \text{UnitPattern}$$

Table 6.4: Product

6.3 Products and function definitions in sequential MPL

To make MPL into a more usable programming language we start introducing some additional features. We start with products and function definitions.

Term formation rules for products are provided in the Table 6.4. One can have a product pattern in the term context (ProductPattern). The product structure pairs terms and allows projection out of a pair. The last two rules analogously concern the unit or empty product.

In MPL, we can define functions we have built; the context acts like a symbol table to remember terms that we have already built. Rules related to context building with function definition and use of functions are in Table 6.5. We can name a term t with the label f (FunctionDef rule). To denote that t is labelled by f we write $(f := t)$.

For example, if we want to use our earlier proved term, $\Lambda AB.\lambda xy.x$ as a function, we can extend the proof:

$$\frac{\Delta \vdash t : T : \alpha \quad \Delta, f : T : \alpha \quad \mathsf{context}}{\Delta, (f := t) : T : \alpha \quad \mathsf{context}}\ \text{FunctionDef}$$

$$\frac{\Delta, (f := t) : T : \alpha \vdash t' : T' : \bullet}{\Delta \vdash (t' \ \texttt{where}\ (f := t) : T : \alpha) : T' : \bullet}\ \text{Where}$$

$$\frac{\Delta, (f := t) : T : \alpha, \Delta' \quad \mathsf{context}}{\Delta, (f := t) : T : \alpha, \Delta' \vdash f : T : \alpha}\ \text{FunctionUse}$$

Table 6.5: Function definition and use

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A:\bullet, B:\bullet, x:A:\bullet, y:B:\bullet \quad \mathsf{context}}\ \text{TermProj}}{A:\bullet, B:\bullet, x:A:\bullet, y:B:\bullet \vdash x:A:\bullet}\ \lambda-\text{abst}}{A:\bullet, B:\bullet, x:A:\bullet \vdash \lambda y.x : B \to A : \bullet}\ \text{TermProj}}{A:\bullet, B:\bullet \vdash \lambda xy.x : A \to B \to A : \bullet}\ \text{TypeAbs}}{A:\bullet \vdash \Lambda B.\lambda xy.x : A \to B \to A : \bullet \to \bullet}\ \text{TypeAbs}}{\vdash \Lambda AB.\lambda xy.x : \Lambda AB.A \to B \to A : \bullet \to \bullet \to \bullet} \qquad f : \Lambda AB.A \to B \to A : \bullet \to \bullet \to \bullet\ \mathsf{ctxt}}{(f := \lambda xy.x) : \Lambda AB.A \to B \to A : \bullet \to \bullet \to \bullet\ \mathsf{ctxt}}$$

We can apply these predefined functions by name, via the FunctionUse rule. Also, we can bind a function using a where clause which would look like where Δ has been defined elsewhere:

$$\frac{\Delta, (f := \lambda xy.x) : \Lambda AB.A \to B \to A : \bullet \to \bullet \to \bullet \vdash t : T : \bullet}{\Delta \vdash (t \ \texttt{where}\ (f := \lambda xy.x) : \Lambda AB.A \to B \to A : \bullet \to \bullet \to \bullet) : T : \bullet}\ \text{Where}$$

6.4 User defined inductive data

Data and codata in MPL provide convenience and flexibility for writing programs. Recall, in the sequential world of MPL in section 3.1 and 3.2, a type is defined using the `(co)data` keyword followed by the choice of rules for constructing or destructing the type. Also, when such a type is declared, the declaration delivers case (or record) and primitve (co)recursion

operators. In order to describe the type system we must give the rules for typing all these operators.

The first rule describes how a new inductive datatype is defined and introduced into the context:

$$\frac{\{\Delta, X_1 : \bullet, \cdots, X_n : \bullet, C : \bullet \vdash F_i\ \widetilde{X_i}\ C : \bullet\}_{i=1\cdots k}}{\Delta,\ \begin{array}{l}\mathsf{data}\ D\ X_1\ \cdots\ X_n \to C = \\ \qquad \{c_i : F_i\ \widetilde{X_i}\ C \to C\}_{i=1\cdots k}\end{array}\ \mathsf{context}}\ \text{DataIntro}$$

Here, $\widetilde{X_i}$ is a sequence of type variables (belonging to the list of type variables $X_1 \cdots X_n$) for c_i.

Example 6.4.1.

(1) The following defines the type of natural numbers:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon\ \ \mathsf{context}}\ \mathsf{new\ C}}{C : \bullet\ \ \mathsf{context}}}{C : \bullet \vdash C : \bullet} \qquad \dfrac{\dfrac{\dfrac{}{\varepsilon\ \ \mathsf{context}}\ \mathsf{new\ C}}{C : \bullet\ \ \mathsf{context}}}{C : \bullet \vdash C : \bullet}}{C : \bullet \vdash C \to C : \bullet}}{\begin{array}{l}\mathsf{data\ Nat} \to \mathsf{C} = \mathsf{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{context} \\ \qquad\qquad\qquad\quad \mathsf{Succ} : \mathsf{C} \to \mathsf{C}\end{array}} \qquad \dfrac{\dfrac{\dfrac{}{\varepsilon\ \ \mathsf{context}}\ \mathsf{new\ C}}{C : \bullet\ \ \mathsf{context}}}{C : \bullet \vdash C : \bullet}$$

(2) Similarly the following defines the type of a `List` for an arbitrary type `A`:

$$\dfrac{\dfrac{\vdots}{A : \bullet, C : \bullet \vdash C : \bullet} \qquad \dfrac{\vdots}{A : \bullet, C : \bullet \vdash A \to C \to C : \bullet}}{\begin{array}{l}\mathsf{data\ List\ A} \to \mathsf{C} = \mathsf{Nil} : \mathsf{C} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{context} \\ \qquad\qquad\qquad\quad \mathsf{Cons} : \mathsf{A} \to \mathsf{C} \to \mathsf{C}\end{array}}$$

MPL also allows mutual inductive data declarations which are natural extensions of the above data declaration. In the sequent, we use $\widetilde{X}$ to denote a list of type variables $X_1 \cdots X_n$ and $\widetilde{X}_{ji}$ (belongs to $\widetilde{X}$) is a sequence of type variables for c_{ji}. The general definition for datatype then looks like:

$$\frac{\{\Delta, \widetilde{X} : \bullet, C_1 : \bullet, \cdots, C_m : \bullet \vdash F_{ji}\ X_{ji}\ C_j : \bullet\}_{i=1\cdots k \text{ and } j=1\cdots m}}{\begin{array}{l}\mathsf{data}\ D_1\ \widetilde{X} \to C_1 = \{c_{1i} : F_{1i}\ \widetilde{X_{1i}}\ C_1 \to C_1\}_{i=1\cdots k} \\ \Delta, \quad \vdots \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{context} \\ \mathsf{and}\ D_m\ \widetilde{X} \to C_m = \{c_{mi} : F_{mi}\ \widetilde{X_{mi}}\ C_m \to C_m\}_{i=1\cdots k}\end{array}}\ \text{DataIntro}$$

Example 6.4.2. The following defines the type of alternating lists:

$$\frac{\begin{array}{cccc}\vdots & \vdots & \vdots & \vdots \\ \Delta \vdash C : \bullet & \Delta \vdash A \to D \to C : \bullet & \Delta \vdash C : \bullet & \Delta \vdash B \to C \to D : \bullet\end{array}}{\begin{array}{l}\mathsf{data\ ListA\ A\ B} \to \mathsf{D} = \mathsf{NilA} : \mathsf{C} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ConsA} : \mathsf{A} \to \mathsf{D} \to \mathsf{C} \qquad \mathsf{context} \\ \mathsf{and\ ListB\ A\ B} \to \mathsf{D} = \mathsf{NilB} : \mathsf{D} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ConsB} : \mathsf{B} \to \mathsf{C} \to \mathsf{D}\end{array}}$$

where, $\Delta = A : \bullet, B : \bullet, C : \bullet, D : \bullet$.

6.4.1 Typing data constructors

The following rule is the typing rule for data constructors:

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash c_{ni} : \Lambda\widetilde{X}.F_{ni}\ \widetilde{X_{ji}}\ (D_i\ \widetilde{X}) : \bullet \to \cdots \to \bullet}\ \text{Cons}$$

where, Δ contains $\begin{array}{l}\mathsf{data}\ D_1\ \widetilde{X} \to C_1 = \{c_{1i} : F_{1i}\ \widetilde{X_{1i}}\ C_1 \to C_1\}_{i=1\cdots k} \\ \vdots \\ \mathsf{and}\ D_m\ \widetilde{X} \to C_m = \{c_{mi} : F_{mi}\ \widetilde{X_{mi}}\ C_m \to C_m\}_{i=1\cdots k}\end{array}$

Example 6.4.3.

(1) In the following example, we derive the type of constructor Succ: assume that the datatype Nat has been defined as above and is in the context, Δ.

$$\frac{\overline{\Delta \quad \mathsf{ctxt}}}{\Delta \vdash \mathsf{Succ} : \mathsf{Nat} \to \mathsf{Nat} : \bullet}\ \text{Cons}$$

(2) In the following example, we derive the type of constructor Cons: we assume that the data List A has been defined as above and is in the context, Δ.

$$\frac{\overline{\Delta \quad \mathsf{ctxt}}}{\Delta \vdash \mathsf{Cons} : \Lambda\, \mathsf{A}.\mathsf{A} \to \mathsf{List\,A} \to \mathsf{List\,A} : \bullet \to \bullet}\ \text{Cons}$$

6.4.2 Typing type constructors

The following rule is the rule for the type constructors introduced by a data declaration:

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash D_i : \bullet \to \cdots \to \bullet}\ \text{TypeCons}$$

$$\mathsf{data}\ D_1\ \widetilde{X} \to C_1 = \{c_{1i} : F_{1i}\ \widetilde{X_{1i}}\ C_1 \to C_1\}_{i=1\cdots k}$$

where, Δ contains $\quad\vdots$

$$\mathsf{and}\ D_m\ \widetilde{X} \to C_m = \{c_{mi} : F_{mi}\ \widetilde{X_{mi}}\ C_m \to C_m\}_{i=1\cdots k}$$

Example 6.4.4.

(1) For example, assume `Nat` is as defined earlier and is in the context, Δ, then the following is legal:

$$\frac{\overline{\Delta \quad \mathsf{ctxt}}}{\Delta \vdash \mathsf{Nat} : \bullet}\ \text{TypCons}$$

i.e. `Nat` is a type.

(2) For example, assume `List A` has been defined earlier and is in the context, Δ, then the following is legal:

$$\frac{\overline{\Delta \quad \mathsf{ctxt}}}{\Delta \vdash \mathsf{List} : \bullet \to \bullet}\ \text{TypCons}$$

6.4.3 Typing the `case` construct

Case allows us to pattern match on the constructors of a datatype. Assume $F_i\ \widetilde{X}\ (D\ \widetilde{X}) = A_1 \to \cdots A_j \to D\ \widetilde{X}$ so $c_i : F_i\ \widetilde{X}\ (D\ \widetilde{X})$. Then the general form of case can be expressed as:

$$\frac{\Delta \vdash t : D\ \widetilde{X} : \bullet \quad \{\Delta, \{\widetilde{x_{ij}} : A_{ij} : \bullet\}_{j=1\cdots n} \vdash t_i : T : \bullet\}_{i=1\cdots m}}{\Delta \vdash \begin{array}{l}\mathsf{case}\ t\ \mathsf{of} \\ \qquad \{c_i\ \widetilde{x_i} \to t_i\}_{i=1\cdots m}\end{array} : T : \bullet}\ \text{case}$$

where $c_1 \cdots c_m$ are the constructors for $D\ \widetilde{X}$. Recall $F_i\ \widetilde{X}\ (D\ \widetilde{X})$ is the type of c_i, the i^{th} constructor of D.

Example 6.4.5. The use of case is shown in the following example that defines the predecessor function, pred:

```
1 pred : Nat -> Nat
2 pred a =
3     case a of
4          Zero ->   Zero
5          Succ n -> n
```

The proof for the body of the function is:

$$\dfrac{\dfrac{\pi_1}{\begin{array}{l}\Delta, a : \mathsf{Nat} : \bullet, () : 1 : \bullet \\ \vdash \mathsf{Zero} : \mathsf{Nat} : \bullet\end{array}} \quad \dfrac{\pi_2}{\begin{array}{l}\Delta, a : \mathsf{Nat} : \bullet, n : \mathsf{Nat} : \bullet \\ \vdash n : \mathsf{Nat} : \bullet\end{array}}}{\Delta, a : \mathsf{Nat} : \bullet \vdash \mathsf{case}\ a\ \mathsf{of}\ \begin{array}{l}\mathsf{Zero} \to \mathsf{Zero} \\ \mathsf{Succ}\ n \to n\end{array} : \mathsf{Nat} : \bullet}\ \text{case}}{\Delta \vdash \lambda a.\ \mathsf{case}\ a\ \mathsf{of}\ \begin{array}{l}\mathsf{Zero} \to \mathsf{Zero} \\ \mathsf{Succ}\ n \to n\end{array} : \mathsf{Nat} \to \mathsf{Nat} : \bullet}\ \lambda - \text{abst}$$

Here, $\Delta = \mathsf{data\ Nat} \to \mathsf{C} = \begin{array}{l} \mathsf{Zero} : \mathsf{C} \\ \mathsf{Succ} : \mathsf{C} \to \mathsf{C} \end{array}$

The sub-proofs denoted as π_1 and π_2 will be completed in Appendix C.

In MPL, we also have case for products. The rule to type check the case for an n-ary product is:

$$\frac{\Delta \vdash t : T_t : \bullet \quad \Delta, (x_1, \cdots, x_n) : T_t : \bullet \vdash t : T : \bullet}{\Delta \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1, \cdots, x_n) \to t : T : \bullet}\ \mathrm{case_p}$$

where, $x_1, \cdots, x_n$ are variables.

Example 6.4.6. A use of $\mathsf{case_p}$ is shown in the following example that defines the swap function, swap:

```
1 -- swapping the elements of a pair
2 swap : (X * Y) -> (Y * X)
3 swap t  =
4      case t of
5           (x , y) -> (y , x)
```

The proof for the body of the function is:

$$\frac{\dfrac{\pi_1}{\begin{array}{l} X : \bullet, Y : \bullet, t : (X \times Y) : \bullet \\ \vdash t : (X \times Y) : \bullet \end{array}} \qquad \dfrac{\pi_2}{\begin{array}{l} X : \bullet, Y : \bullet, t : (X \times Y) : \bullet, (x, y) : (X \times Y) : \bullet \\ \vdash (y, x) : (Y \times X) : \bullet \end{array}}}{X : \bullet, Y : \bullet, t : (X \times Y) : \bullet \vdash \mathsf{case}\ t\ \mathsf{of}\ (x, y) \to (y, x)\ : (Y \times X) : \bullet}\ \mathrm{case_p}$$

The completed proof will be given in Appendix C.

6.4.4 The `fold` operation

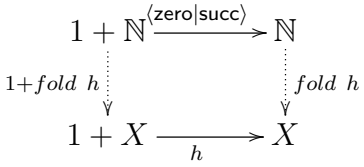
The fold construct allows one to write programs which iterate over a datatype. In [**?**], it is described that datatypes come packaged with a primitive recursion operator which we explain below.

6.4.4.1 Initial algebras

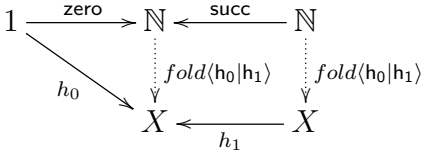
From a categorical perspective, a datatype is an initial algebra, $\bar{F}$, for a functor $F : \mathbb{X} \to \mathbb{X}$. An algebra for F with carrier X is given by a map: $h : F(X) \to X$, called the structure map of the algebra. The structure map of the initial algebra is called the constructor of the datatype and written $\mathsf{cons_F} : F(\bar{F}) \to \bar{F}$. Being an initial algebra means that given any algebra for $h : F(X) \to X$, there is a unique map, $fold\ h$, that makes the following square commute:

$$\begin{array}{ccc} F(\bar{F}) & \xrightarrow{cons} & \bar{F} \\ {\scriptstyle F(fold\ h)}\downarrow & & \downarrow{\scriptstyle fold\ h} \\ F(X) & \xrightarrow[h]{} & X \end{array}$$

When the functor F is a coproduct (sum) of functors, then cons can be canonically viewed as a copairing of constructors – one for each component of the sum of functors. Consider in any category $\mathbb{X}$ with a final object, 1, and coproducts, $F : \mathbb{X} \to \mathbb{X} : X \to 1 + X$, the functor $F(X) = 1 + X$, with $F(f) = 1 + f$, an algebra is of the form $h : 1 + X \to X$. Now by universality, a map out of a coproduct is always a choice of maps. Thus h breaks into a choice of maps $h = \langle h_0 \mid h_1 \rangle$, where $h_0 : 1 \to X$ (i.e. $z \in X$) and $h_1 : X \to X$. The initial algebra of this functor, which is a natural number object, $\mathbb{N}$, has constructor $\mathsf{cons} : 1 + \mathbb{N} \to \mathbb{N}$, where $\mathsf{cons} = \langle \mathsf{zero} \mid \mathsf{succ} \rangle : 1 + \mathbb{N} \to \mathbb{N}$ with, $\mathsf{zero} : 1 \to \mathbb{N}$ and $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. One may think of $\mathbb{N}$ as being freely generated by zero and succ. The datatype of the natural numbers is defined as the type satisfying for any $h : 1 + X \to X$ the universal diagram:



Letting $h = \langle h_0 \mid h_1 \rangle$ we may rewrite this universal property



Thus,

$$\mathsf{zero}(\mathsf{fold}\ \langle h_0 \mid h_1 \rangle) = h_0$$

$$\mathsf{succ}(\mathsf{fold}\ \langle h_0 \mid h_1 \rangle)(x) = (\mathsf{fold}\ \langle h_0 \mid h_1 \rangle) h_1(x)$$

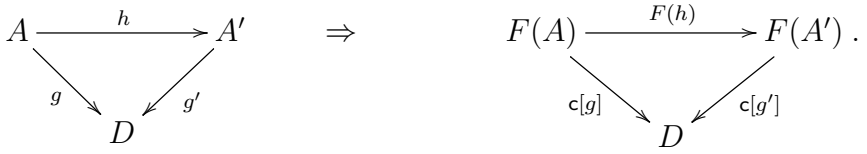
Thus we recover a form of primitive recursion on the natural number object, $\mathbb{N}$.

We could use the initial algebra description of datatypes directly to provide type recursion; however we shall use different but more convenient form which is equivalent, which we now introduce.

6.4.4.2 Circular definitions

There is an alternative way of defining the universal property of a datatype using a **circular combinator** [**?**]. We will review this circular definition of datatypes and how they fit into the syntax and type system of MPL. For further information on these datatypes, see [**?**, **?**].

The basic structure involved in defining circular datatypes is called a circular combinator. A **circular combinator** over D for a functor F is a way to produce from a map $g : A \to D$, a map $c[g] : F(A) \to D$, such that

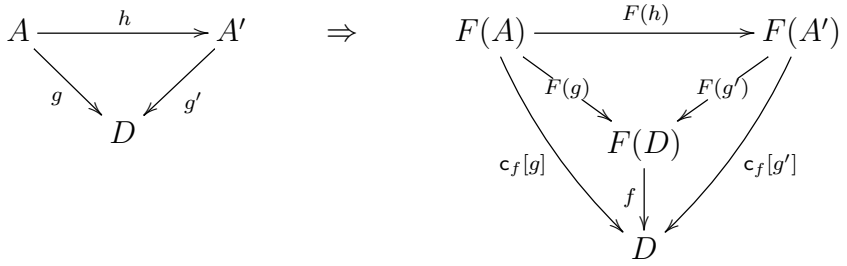


Lemma 1. *(See [?, ?]) Circular combinators for a functor $F : \mathbb{X} \to \mathbb{X}$ over D correspond precisely to F-algebras on D. Given an algebra $f : F(D) \longrightarrow D$ one obtains a circular combinator, $\mathsf{c}_f[g] := F(g)f$ and conversely, given a circular combinator $\mathsf{c}[\_]$ one obtains an algebra, $\mathsf{c}[1_D] : F(D) \longrightarrow D$.*

Proof. If $f : F(D) \longrightarrow D$ is an algebra, then a circular combinator, $\mathsf{c}_f[g]$ is defined as follows:

$$\frac{A \xrightarrow{g} D}{F(A) \xrightarrow{F(g)} F(D) \xrightarrow{f} D}\ \mathsf{c}_f$$

So $\mathsf{c}_f[g] = F(g)f$ and the implication

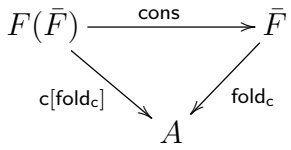


clearly holds. Conversely, a circular combinator, $\mathsf{c}[\_]$ defines an algebra by applying it to the identity map:

$$\frac{D \xrightarrow{1} D}{F(D) \xrightarrow{\mathsf{c}[1_D]} D}\ \mathsf{c}$$

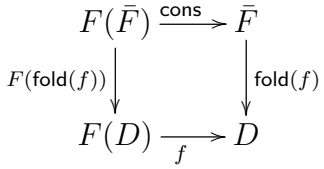
Clearly $c_f[1_D] = f$ and conversely $\mathsf{c}_f[g] = F(g)c_f[1_D]$: so circular combinator and algebras are in bijective correspondence. □

To obtain the circular definition of a datatype for a functor, F, we need an object $\bar{F}$ and a map $\mathsf{cons} : F(\bar{F}) \to \bar{F}$ such that, for every combinator $c[\_]$ there is a unique map $\bar{F} \xrightarrow{\mathsf{fold_c}} A$ such that

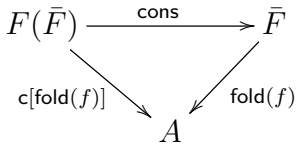


Proposition 6.4.1. *(See [?, ?]) The algebraic definition and the circular definition of datatypes are equivalent.*

Proof. Given $\bar{F}$ satisfying the algebraic definition of an inductive datatype, we must show it satisfies the circular definition of the datatype. If $\bar{F}$ is the initial algebra, then for any algebra $f : F(D) \to D$ there is a unique $fold(f)$ such that the following diagram commutes.

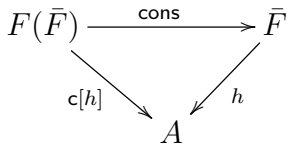


Given a circular combinator $\mathsf{c}[\_]$ of F over D one may extract an algebra $f = \mathsf{c}[1_D] : F(D) \to D$. Then we set $fold_c = \mathsf{fold}(f)$ and the following circular diagram:



commutes as $F(\mathsf{fold}(f)) = \mathsf{c}_f[\mathsf{fold}(f)]$ and $\mathsf{fold}(f)$ is certainly unique. Thus, $\mathsf{fold}(f)$ satisfies the properties required by the circular map $fold_c$.

Conversely, if the circular definition of an inductive datatype holds for $\bar{F}$, then we want to show that it must satisfy the algebra definition. Given an algebra $f : F(D) \to D$ one can define a circular combinator $\mathsf{c}_f[\_]$ such that $\mathsf{c}_f[g] = F(g)f$ then $\exists!\, h$



□

Again, we consider, $\mathbb{N}$, the datatype generated freely by $1+\mathbb{N} \to \mathbb{N}$ i.e, $\langle \mathsf{zero} \mid \mathsf{succ} \rangle$ freely generated by $1 \xrightarrow{zero} \mathbb{N}$ and $\mathbb{N} \xrightarrow{succ} \mathbb{N}$.

Any combinator for the functor $F(X) = 1 + X$ is of the form:

$$
\begin{aligned}
\text{c[f]}: \quad & 1 + \mathbf{X} \xrightarrow{1+f} 1 + A \xrightarrow{[z,\,s]} A \\
& = 1 + \mathbf{X} \xrightarrow{[z,\,fs]} A \\
& = \left(\begin{array}{l} case\ m\ of \\ \quad () \mapsto zero \\ \quad x \mapsto succ(f\ x) \end{array} \right) : 1 + \mathbf{X} \to A
\end{aligned}
$$

The universal property we have is then for all $c[\_]$, there is a unique $fold_c$. This tells us again that

$$\mathsf{fold_c}\ \mathsf{zero} = z \qquad \text{and} \qquad \mathsf{fold_c}\ (\mathsf{succ}\ n) = s\ (\mathsf{fold_c}\ n)$$

i.e., we again recover the primitive recursion equations described in the previous section.

In MPL the syntax for defining $f : \mathbb{N} \to A$ as a fold is:

```
fold
    f : N -> A
    f x by x as
      Zero -> z
```

```
Succ a -> s (f a)
```

If f is to be written as a fold it suffices to provide the combinator which is

```
() -> z
a -> s (f a) .
```

This is a combinator (with the case implicit) that we need to type check. Since a combinator, $c[f] = [z, fs]$ is defined over all f, we use any f in the context with right type. We thus have:

$$\frac{\Delta, C : \bullet, f : C \to A : \bullet \vdash t_1 : A \quad \Delta, C : \bullet, f : C \to A : \bullet, a : C : \bullet \vdash t_2 : A : \bullet}{\begin{array}{l}\mathsf{fold} \\ \Delta, \quad \begin{array}{l} f : \mathbb{N} \to A \\ f\ x\ \mathsf{by}\ x\ \mathsf{as} \\ \quad \mathsf{Zero} \to t_1 \\ \quad \mathsf{Succ}\ a \to t_2 \end{array} \quad \mathsf{context}\end{array}}\ \text{fold}$$

This generalizes into the following typechecking rule:

$$\frac{\{\Delta, C : \bullet, f : C \to Z : \bullet, \{\widetilde{p_{ij}} : A_{ij} : \bullet\}_{j=1,\cdots,n} \vdash t_i : Z : \bullet\}_i}{\begin{array}{l}\mathsf{fold} \\ \Delta, \quad \begin{array}{l} f : D\ \widetilde{X} \to Z \\ f\ x\ \mathsf{by}\ x\ \mathsf{as} \\ \quad c_i\ \widetilde{p_i} \to t_i \end{array} \quad \mathsf{context}\end{array}}\ \text{fold}$$

where c_i are the constructors for a datatype defined in Δ. Recall $F_i\ \widetilde{X}\ D\ \widetilde{X}$ is the type of c_i, the ith constructor of D and $F_i\ \widetilde{X}\ D\ \widetilde{X} = A_1 \to \cdots A_j \to C$.

Example 6.4.7. Here an example of how to typecheck the recursive function add : Nat $\to$ Nat $\to$ Nat, is given. The sub-proofs of this proof are denoted as π_1 and π_2, and will be completed in Appendix C.

$$\dfrac{\dfrac{\pi_1}{\begin{array}{l}\Delta, X : \bullet, add : X \to (\mathsf{Nat} \to \mathsf{Nat}) : \bullet, \\ x : \mathsf{Nat} : \bullet, y : \mathsf{Nat} : \bullet \vdash y : \mathsf{Nat} : \bullet\end{array}} \qquad \dfrac{\pi_2}{\begin{array}{l}\Delta, X : \bullet, add : X \to (\mathsf{Nat} \to \mathsf{Nat}) : \bullet, \\ x : \mathsf{Nat} : \bullet, y : \mathsf{Nat} : \bullet, n : X : \bullet \\ \vdash \mathsf{Succ}\ (add\ n\ y) : \mathsf{Nat} : \bullet\end{array}}}{\begin{array}{l}\mathsf{fold} \\ \quad add : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \Delta, \quad add\ x\ y\ by\ x\ \mathsf{as} \qquad\qquad \mathsf{context} \\ \quad\ \mathsf{Zero} \to y \\ \quad\ \mathsf{Succ}\ n \to \mathsf{Succ}\ (add\ n\ y)\end{array}}\ \text{fold}$$

Here is the rule for the mutual fold combinator in MPL:

$$\dfrac{\{\Delta, \widetilde{C} : \bullet, f_i : C_p \to Z_p : \bullet, \{\widetilde{p_{ij}} : A_{ij} : \bullet\}_{j=1,\cdots,n} \vdash t_{ji} : Z_j : \bullet\}_i}{\begin{array}{l}\mathsf{fold} \\ \quad f_1 : D_1\ \widetilde{X} \to Z_1 \\ \quad f_1\ \widetilde{x}\ by\ x_{1k}\ \mathsf{as} \\ \quad\ c_{1i}\ \widetilde{p_{1i}} \to t_{1i} \\ \Delta, \quad\ \vdots \qquad\qquad \mathsf{context} \\ \quad f_n : D_n\ \widetilde{X} \to Z_n \\ \quad f_n\ \widetilde{x}\ by\ x_{nk}\ \mathsf{as} \\ \quad\ c_{ni}\ \widetilde{p_{ni}} \to t_{ni}\end{array}}\ \text{fold}$$

Example 6.4.8. In the following example, we use mutual recursion on the non-mutually recursive datatype, $\mathbb{N}$, to determine whether a natrual number is even or odd:

$$\frac{\begin{array}{cccc}\pi_1 & \pi_2 & \pi_3 & \pi_4\\ \Delta, \Delta', () : 1 : \bullet & \Delta, \Delta', n_1 : X : \bullet & \Delta, \Delta', () : 1 : \bullet & \Delta, \Delta', n_2 : X : \bullet \\ \vdash \mathsf{True} : \mathbb{B} & \vdash odd(n_1) : \mathbb{B} & \vdash \mathsf{True} : \mathbb{B} & \vdash even(n_2) : \mathbb{B}\end{array}}{\begin{array}{l}fold \\ \quad even : \mathbb{N} \to \mathbb{B} \\ \quad even\ x\ by\ x\ as \\ \quad\quad \mathsf{Zero} \mapsto \mathsf{True} \\ \Delta, \quad\quad \mathsf{Succ}\ n_1 \mapsto odd(n_1) \quad \mathsf{context} \\ \quad odd : \mathbb{N} \to \mathbb{B} \\ \quad odd\ x\ by\ x\ as \\ \quad\quad \mathsf{Zero} \mapsto \mathsf{False} \\ \quad\quad \mathsf{Succ}\ n_2 \mapsto even(n_2)\end{array}} \text{ fold}$$

$$\Delta = \begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathsf{Zero} : \mathsf{C} \\ \qquad\qquad\quad \mathsf{Succ} : \mathsf{C} \to \mathsf{C}\end{array}, \quad \begin{array}{l}\mathsf{data}\ \mathbb{B} \to \mathsf{C} = \mathsf{True} : \mathsf{C} \\ \qquad\qquad\quad\ \mathsf{False} : \mathsf{C}\end{array}$$

$\Delta' = X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}$

The complete proof is given in Appendix C.

We collect all the rules related to user defined inductive data in one Table 6.6.

6.5 User defined coinductive data

The rules for defining and using codata are introduced in this section. As for data, we can put codata definitions in the context. Any codata definition in context delivers a type constructor (i.e. the name of the codatatype with arguments), data destructors, records and an unfold operation. The type system must have the rules for typing all these constructs.

The first rule describes how a new codatatype is defined and introduced into the context:

$$\frac{\{\Delta, \widetilde{X} : \bullet, C_1 : \bullet, \cdots, C_m : \bullet \vdash F_{ji}\ X_{ji}\ C_j : \bullet\}_{i=1\cdots k \text{ and } j=1\cdots m}}{\begin{array}{l}\mathsf{data}\ D_1\ \widetilde{X} \to C_1 = \{c_{1i} : F_{1i}\ \widetilde{X_{1i}}\ C_1 \to C_1\}_{i=1\cdots k} \\ \Delta, \quad \vdots \qquad\qquad \mathsf{context} \\ \mathsf{and}\ D_m\ \widetilde{X} \to C_m = \{c_{mi} : F_{mi}\ \widetilde{X_{mi}}\ C_m \to C_m\}_{i=1\cdots k}\end{array}}\ \text{DataIntro}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash c_{ni} : \Lambda\widetilde{X}.F_{ni}\ \widetilde{X_{ji}}\ (D_i\ \widetilde{X}) : \bullet \to \cdots \to \bullet}\ \text{Cons}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash D_i : \bullet \to \cdots \to \bullet}\ \text{TypeCons}$$

$$\frac{\Delta \vdash t : D\ \widetilde{X} : \bullet \quad \{\Delta, \{\widetilde{x_{ij}} : A_{ij} : \bullet\}_{j=1\cdots n} \vdash t_i : T : \bullet\}_{i=1\cdots m}}{\Delta \vdash \begin{array}{l}\mathsf{case}\ t\ \mathsf{of} \\ \quad \{c_i\ \widetilde{x_i} \to t_i\}_{i=1\cdots m}\end{array} : T : \bullet}\ \text{case}$$

$$\frac{\Delta \vdash t : T_t : \bullet \quad \Delta, (x_1, \cdots, x_n) : T_t : \bullet \vdash t : T : \bullet}{\Delta \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1, \cdots, x_n) \to t : T : \bullet}\ \text{case}_\text{p}$$

$$\frac{\{\Delta, \widetilde{C} : \bullet, f_i : C_p \to Z_p : \bullet, \{\widetilde{p_{ij}} : A_{ij} : \bullet\}_{j=1,\cdots,n} \vdash t_{ji} : Z_j : \bullet\}_i}{\begin{array}{l}\mathsf{fold} \\ \quad f_1 : D_1\ \widetilde{X} \to Z_1 \\ \quad f_1\ \widetilde{x}\ by\ x_{1k}\ \mathsf{as} \\ \quad\quad c_{1i}\ \widetilde{p_{1i}} \to t_{1i} \\ \Delta, \quad\quad \vdots \qquad\qquad \mathsf{context} \\ \quad f_n : D_n\ \widetilde{X} \to Z_n \\ \quad f_n\ \widetilde{x}\ by\ x_{nk}\ \mathsf{as} \\ \quad\quad c_{ni}\ \widetilde{p_{ni}} \to t_{ni}\end{array}}\ \text{fold}$$

Table 6.6: Rules related to inductive data

$$\frac{\{\Delta, \widetilde{X} : \bullet, C_1 : \bullet, \cdots, C_m : \bullet \vdash G_{ji}\ \widetilde{X_{ji}}\ C_j : \bullet\}_{i=1\cdots k\ and\ j=1\cdots m}}{\begin{array}{l}\Delta, \\ \quad \vdots \quad \text{context} \\ \text{codata } C_1 \to E_1\ \widetilde{X} = \{d_{1i} : C_1 \to G_{1i}\ \widetilde{X_{1i}}\ C_1\}_{i=1,\cdots,k} \\ \text{and } C_m \to E_n\ \widetilde{X} = \{d_{mi} : C_m \to G_{ni}\ \widetilde{X_{ni}}\ C_m\}_{i=1,\cdots,k}\end{array}}\ \text{CoDataIntro}$$

Example 6.5.1.

(1) The following defines the type of infinite list of a type:

$$\frac{\begin{array}{c}\vdots \\ \mathsf{A}:\bullet, \mathsf{C}:\bullet \vdash \mathsf{C} \to \mathsf{A}:\bullet\end{array} \quad \begin{array}{c}\vdots \\ \mathsf{A}:\bullet, \mathsf{C}:\bullet \vdash \mathsf{C} \to \mathsf{C}:\bullet\end{array}}{\begin{array}{l}\mathsf{codata\ C} \to \mathsf{InfList\ A} = \mathsf{Head} : \mathsf{C} \to \mathsf{A} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Tail} : \mathsf{C} \to \mathsf{C}\end{array}\ \text{context}}$$

(2) The following defines the type of a mutual recursive codata for a game:

$$\frac{\begin{array}{c}\vdots \\ \Delta, \mathsf{A}:\bullet, \mathsf{B}:\bullet, \mathsf{C}:\bullet, \mathsf{D}:\bullet \vdash \mathsf{C} \to \mathsf{List\ (A,\ D)}:\bullet\end{array} \quad \begin{array}{c}\vdots \\ \Delta, \mathsf{A}:\bullet, \mathsf{B}:\bullet, \mathsf{C}:\bullet, \mathsf{D}:\bullet \vdash \mathsf{D} \to \mathsf{List\ (B,\ C)}:\bullet\end{array}}{\begin{array}{l}\Delta, \\ \quad \begin{array}{l}\mathsf{codata\ C} \to \mathsf{PGame\ A\ B} = \mathsf{Pmove} : \mathsf{C} \to \mathsf{List\ (A,\ D)} \\ \mathsf{and\ D} \to \mathsf{OGame\ A\ B} = \mathsf{Omove} : \mathsf{D} \to \mathsf{List\ (B,\ C)}\end{array}\ \text{context}\end{array}}$$

where Δ contains List datatype.

6.5.1 Typing codata destructors

The following rule is the typing rule for codata destructors:

$$\frac{\Delta \quad \text{context}}{\Delta \vdash d_{ni} : \Lambda\widetilde{X}.G_{ni}\ \widetilde{X_{ji}}\ (E_i\ \widetilde{X}) : \bullet \to \cdots \to \bullet}\ \text{Des}$$

where, Δ contains
$$\begin{array}{l}\text{codata } C_1 \to E_1\ \widetilde{X} = \{d_{1i} : C_1 \to G_{1i}\ \widetilde{X_{1i}}\ C_1\}_{i=1,\cdots,k} \\ \quad \vdots \\ \text{and } C_m \to E_n\ \widetilde{X} = \{d_{mi} : C_m \to G_{ni}\ \widetilde{X_{ni}}\ C_m\}_{i=1,\cdots,k}\end{array}$$

Example 6.5.2.

(1) One of the codata destructors for the infinite list of natural numbers, Head, is given below where data InfList Nat is available in the context:

$$\frac{\overline{\Delta \quad \mathsf{ctxt}}}{\Delta \vdash \mathsf{Head} : \Lambda \mathsf{A}.\mathsf{InfList\ A} \to \mathsf{A} : \bullet \to \bullet}\ \text{Des}$$

(2) One of the codata destructors for the previously discussed mutual recursive codata, PMove, is given below where codata PGame A B and OGame A B is available in the context:

$$\frac{\overline{\Delta \quad \mathsf{ctxt}}}{\Delta \vdash \mathsf{Pmove} : \Lambda \mathsf{AB}.\mathsf{PGame\ A\ B} \to \mathsf{A} \to \mathsf{SF\ OGame\ A\ B} : \bullet \to \bullet \to \bullet}\ \text{Des}$$

6.5.2 Typing type constructors

The following rule is the typing rule for codatatype projection from the context:

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash E_i : \bullet \to \cdots \to \bullet}\ \text{TypeDes}$$

where, Δ contains

$$\begin{array}{l} \mathsf{codata}\ C_1 \to E_1\ \widetilde{X} = \{d_{1i} : C_1 \to G_{1i}\ \widetilde{X_{1i}}\ C_1\}_{i=1,\cdots,k} \\ \vdots \\ \mathsf{and}\ C_m \to E_n\ \widetilde{X} = \{d_{mi} : C_m \to G_{ni}\ \widetilde{X_{ni}}\ C_m\}_{i=1,\cdots,k} \end{array}$$

Example 6.5.3. The type constructor for InfList A is typed as follows:

$$\frac{\overline{\Delta \quad \mathsf{ctxt}}}{\Delta \vdash \mathsf{InfList} : \bullet \to \bullet}\ \text{TypDes}$$

assuming datatype InfList A is in the context.

6.5.3 Typing the `record` construct

The typing rule for record is:

$$\frac{\{\Delta \vdash t_i : G_i\ \widetilde{X}\ (E_i\ \widetilde{X}) : \bullet\}_{i=1\cdots m}}{\Delta \vdash \begin{array}{c}\mathsf{record} \\ \\ \{d_i \leftarrow t_i\}_{i=1\cdots m}\end{array} : T : \bullet}\ \text{record}$$

Example 6.5.4. Consider the following example:

```
1 codata t -> Triple a b c = First : t -> a
2                            Second : t -> b
3                            Third : t -> c
4 makeTriple : a -> b -> c -> Triple a b c
5 makeTriple a b c =
6     record First  <- a
7            Second <- b
8            Third  <- c
```

$$\frac{\begin{array}{ccc}\vdots & \vdots & \vdots \\ \Delta \vdash a : A : \bullet & \Delta \vdash b : B : \bullet & \Delta \vdash c : C : \bullet\end{array}}{\begin{array}{c}\mathsf{record} \\ \\ \Delta \vdash \end{array}\begin{array}{l}\mathsf{First} \leftarrow a \\ \\ \mathsf{Second} \leftarrow b \\ \\ \mathsf{Third} \leftarrow c\end{array} : A \to B \to C \to \mathsf{Triple}\ A\ B\ C : \bullet}\ \text{record}$$

where,

$$\Delta = \begin{array}{l}\mathsf{codata}\ t \to \mathsf{Triple}\ a\ b\ c = \\ \\ \quad \mathsf{First} : t \to a \\ \\ \quad \mathsf{Second} : t \to b \\ \\ \quad \mathsf{Third} : t \to c\end{array} , A : \bullet, B : \bullet, C : \bullet, a : A : \bullet, b : B : \bullet, c : C : \bullet$$

6.5.4 Typing `unfold`

Here is the typing rule for the unfold combinator in MPL:

$$\frac{\{\Delta, \widetilde{C} : \bullet, f_i : Z_p \to C_p : \bullet \vdash t_{ji} : G_i\ \widetilde{X}\ (E_i\ \widetilde{X}) : \bullet\}_i}{\Delta, \begin{array}{l}\textsf{unfold}\\ \quad f_1 : Z_1 \to (E_i\ \widetilde{X})\\ \quad f_1\ \widetilde{x} =\\ \quad\quad d_{1i} \leftarrow t_{1i}\\ \quad\quad \vdots\\ \quad f_n : Z_n \to (E_i\ \widetilde{X})\\ \quad f_n\ \widetilde{x} =\\ \quad\quad d_{ni} \leftarrow t_{ni}\end{array} \textsf{context}}\ \text{unfold}$$

Example 6.5.5. We type the example function nats that represents the infinite list; $0, 1, 2, \cdots$. The unfold gives nats function which provides that infinite list.

$$\frac{\begin{array}{cc}\vdots & \vdots\\ \Delta, X : \bullet, nats : X : \bullet, n : \textsf{InfList Nat} : \bullet & \Delta, X : \bullet, nats : X : \bullet, n : \textsf{InfList Nat} : \bullet\\ \vdash n : \textsf{Nat} : \bullet & \vdash nats\ (\textsf{Succ}\ m) : \textsf{InfList Nat} : \bullet\end{array}}{\Delta, \begin{array}{l}\textsf{unfold}\\ \quad nats : \textsf{InfList Nat}\\ \quad nats\ n :=\\ \quad\quad \textsf{Head} \leftarrow n\\ \quad\quad \textsf{Tail} \leftarrow nats\ (\textsf{Succ}\ m)\end{array} \textsf{context}}\ \text{unfold}$$

assuming natural number data, Nat and infinite list codata, InfList Nat in the context.

We can collect all the rules related to user defined inductive data in one Table 6.7.

$$\frac{\{\Delta, \widetilde{X} : \bullet, C_1 : \bullet, \cdots, C_m : \bullet \vdash G_{ji}\ \widetilde{X_{ji}}\ C_j : \bullet\}_{i=1\cdots k\ and\ j=1\cdots m}}{\begin{array}{l}\mathsf{codata}\ C_1 \to E_1\ \widetilde{X} = \{d_{1i} : C_1 \to G_{1i}\ \widetilde{X_{1i}}\ C_1\}_{i=1,\cdots,k} \\ \Delta, \quad \vdots \qquad\qquad\qquad\qquad \mathsf{context} \\ \mathsf{and}\ C_m \to E_n\ \widetilde{X} = \{d_{mi} : C_m \to G_{ni}\ \widetilde{X_{ni}}\ C_m\}_{i=1,\cdots,k}\end{array}}\ \text{CoDataIntro}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash d_{ni} : \Lambda\widetilde{X}.G_{ni}\ \widetilde{X_{ji}}\ (E_i\ \widetilde{X}) : \bullet \to \cdots \to \bullet}\ \text{Des}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash E_i : \bullet \to \cdots \to \bullet}\ \text{TypeDes}$$

$$\frac{\{\Delta \vdash t_i : G_i\ \widetilde{X}\ (E_i\ \widetilde{X}) : \bullet\}_{i=1\cdots m}}{\Delta \vdash \begin{array}{l}\mathsf{record} \\ \quad \{d_i \leftarrow t_i\}_{i=1\cdots m}\end{array} : T : \bullet}\ \text{record}$$

$$\frac{\{\Delta, \widetilde{C} : \bullet, f_i : Z_p \to C_p : \bullet \vdash t_{ji} : G_i\ \widetilde{X}\ (E_i\ \widetilde{X}) : \bullet\}_i}{\begin{array}{l}\mathsf{unfold} \\ \quad f_1 : Z_1 \to (E_i\ \widetilde{X}) \\ \quad f_1\ \widetilde{x} = \\ \quad\quad d_{1i} \leftarrow t_{1i} \\ \Delta, \quad\quad \vdots \qquad\qquad \mathsf{context} \\ \quad f_n : Z_n \to (E_i\ \widetilde{X}) \\ \quad f_n\ \widetilde{x} = \\ \quad\quad d_{ni} \leftarrow t_{ni}\end{array}}\ \text{unfold}$$

Table 6.7: Rules related to coinductive data

Chapter 7

The Type System for the Concurrent world

In this chapter, the type system for the concurrent world will be described. The type system shows, in detail, how the type of a process is determined by its syntactic structure. This provides the basis for type checking and type inference.

7.1 Type system for concurrent MPL

The kinds for concurrent MPL are generated by:

$$\alpha := \circ \mid \bullet \to \alpha \mid \circ \to \alpha$$

where, $\circ$ is the most basic kind. These allow for concurrent programs to be parametric on both concurrent types (or protocols) and sequential types.

In the concurrent world, there are many sorts of terms and types which can be produced and we start by giving a brief overview of these to help the reader.

(1) *Protocols*: These are the basic types of the concurrent world. A protocol T has kind $\circ$ which we indicate by writing $T : \circ$.

(2) *Process types*: These take the form

$$(\Phi \Rightarrow \Psi) :: \circ$$

where Φ is a list of protocols which have an "input polarity" and Ψ is a list of protocols of "output polarity".

(3) *Process types with channel names*: These are denoted by

$$(\alpha \Rightarrow \beta) \triangleright (\Phi \Rightarrow \Psi) \blacktriangleleft \circ.$$

Here $\Phi \Rightarrow \Psi$ is a process type and $\alpha \Rightarrow \beta$ is a pair of corresponding lists of channel names, which allow us to refer to channels by name rather than by their position in $\Phi \Rightarrow \Psi$. We shall regard the following two forms

$$(\alpha_1, \alpha_2 \Rightarrow \beta_1, \beta_2) \triangleright (T_1, T_2 \Rightarrow S_1, S_2) \blacktriangleleft \circ$$

and

$$(\alpha_1 \triangleright T_1, \alpha_2 \triangleright T_2) \Rightarrow (\beta_1 \triangleright S_1, \beta_2 \triangleright S_2) \blacktriangleleft \circ$$

to be equivalent. The former form is particularly convenient when one wants to express the step of abstracting the channel names (which is necessary to obtain a process).

(4) *Process terms*: These can only be built once the channel names have been introduced (as above). They have the form:

$$t \blacktriangleleft (\alpha \Rightarrow \beta) \triangleright (\Phi \Rightarrow \Psi) \blacktriangleleft \circ.$$

(5) *Processes*: Once one has built a term using channel names, one can abstract the channel names away to obtain a process which can be used with any channel names. A process Q is then written as $Q :: \Phi \Rightarrow \Psi :: \circ$ where $\Phi \Rightarrow \Psi$ is its process type.

A **term in the context** is a sequent of the form:

$$\Delta \vdash P$$

where P is a single formula which can be any of the items discussed above. The context, Δ, should be regarded as a symbol table which contains the variables, functions, types and other terms required to build a process. Every process which can be derived in this system can be added into its context as a definition; this allows one to build contexts with predefined processes.

7.1.1 Basic protocols

For the process world, the basic kind for a protocol is $\circ$. The basic protocols we are going to introduce are built from the protocol constructors $\top, \bot, \otimes_{\_}, \oplus_{\_}, \gg_{\_}, \ll_{\_}$. Table 7.1 shows the rules for building these basic types, called protocols, in the process world.

7.1.2 Process types and basic terms

In Table 7.2, the rule ProcessTypeIntro, builds a process type, which is essentially a two-sided sequent of protocols. It can be created from the protocols introduced in the Table 7.1. The first three rules of this table are for building process types. There are two types of abstraction in the process world. The rule ProcessTermAbs, allows a sequential term to be abstracted over a concurrent term while the rule ProcessProcessAbs, allows a process to be abstracted over another process. This allows a process to take other processes and sequential terms as arguments. We can also discharge these abstractions by applying them respectively to a sequential term (TermApp rule) or a concurrent term (ProcessApp rule).

7.1.3 Typing process types with channels

Table 7.3 shows the relation between channel names and process types. We can assign channel names to the protocols in process types by using ChannelNaming. When a process type, $\Phi \Rightarrow \Psi :: \alpha$, is valid, where, Φ and Ψ are list of protocols and α is a kind, can attach channel names $\alpha \Rightarrow \beta$ to the protocols. When channel names have been attached to the protocols, we use the symbol, $\blacktriangleleft$ to indicate that it is not a pure process and that channel names are being used. These are therefore called processes with channel names. The channel names in the process types can be abstracted always by using the ChannelAbs rule to obtain a process. The rule for calling a predefined process (the call rule) requires channel names to be added before we can use a predefined process in a process term.

$$\frac{\Delta \ \ \mathsf{context} \quad new\ T}{\Delta, T : \circ \qquad \mathsf{context}}\ \text{TypeVarIntro}$$

$$\frac{\Delta, T : \circ, \Delta' \ \ \mathsf{context}}{\Delta, T : \circ, \Delta' \vdash T : \circ}\ \text{TypeProj}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash \bot : \circ}\ \text{Bot}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash \top : \circ}\ \text{Top}$$

$$\frac{\Delta \vdash T : \circ \ \ \Delta \vdash T' : \circ}{\Delta \vdash (T \otimes T') : \circ}\ \text{TypeTensor}$$

$$\frac{\Delta \vdash T : \circ \ \ \Delta \vdash T' : \circ}{\Delta \vdash (T \oplus T') : \circ}\ \text{TypePar}$$

$$\frac{\Delta \vdash T : \bullet \ \ \Delta \vdash T' : \circ}{\Delta \vdash (T \ll T') : \circ}\ \text{TypeGet}$$

$$\frac{\Delta \vdash T : \bullet \ \ \Delta \vdash T' : \circ}{\Delta \vdash (T \gg T') : \circ}\ \text{TypePut}$$

$$\frac{\Delta \vdash T : \circ}{\Delta \vdash neg\ (T) : \circ}\ \text{TypeNeg}$$

Table 7.1: Basic protocols

$$\frac{\{\Delta \vdash T : \circ\}_{T \in \Phi, \Psi}}{\Delta \vdash \Phi \Rightarrow \Psi :: \circ} \text{ ProcessTypeIntro}$$

$$\frac{\Delta \vdash \mathcal{T} :: \circ \quad \Delta \vdash T : \bullet}{\Delta \vdash T \to \mathcal{T} :: \circ} \text{ PTypeTypeAbs}$$

$$\frac{\Delta \vdash \mathcal{T} :: \circ \quad \Delta \vdash \mathcal{S} :: \circ}{\Delta \vdash \mathcal{T} \twoheadrightarrow \mathcal{S} :: \circ} \text{ PTypePTypeAbs}$$

$$\frac{\Delta \vdash \mathcal{T} :: \circ \quad \mathit{new}\ Q}{\Delta, Q :: \mathcal{T} :: \circ \quad \mathsf{context}} \text{ ProcessName}$$

$$\frac{\Delta, Q :: \mathcal{T} :: \circ, \Delta' \quad \mathsf{context}}{\Delta, Q :: \mathcal{T} :: \circ, \Delta' \vdash Q :: \mathcal{T} :: \circ} \text{ ProcessProj}$$

$$\frac{\Delta, x : T : \bullet \vdash t :: \mathcal{T} :: \circ}{\Delta \vdash \lambda x.t :: T \to \mathcal{T} :: \circ} \text{ ProcessTermAbs}$$

$$\frac{\Delta, P :: \mathcal{T} :: \circ \vdash Q :: \mathcal{S} :: \circ}{\Delta \vdash \lambda P.Q :: \mathcal{T} \twoheadrightarrow \mathcal{S} :: \circ} \text{ ProcessProcessAbs}$$

$$\frac{\Delta \vdash P :: T \to \mathcal{T} :: \circ \quad \Delta \vdash t : T : \bullet}{\Delta \vdash P(t) :: \mathcal{T} :: \circ} \text{ TermApp}$$

$$\frac{\Delta \vdash P :: \mathcal{T}' \to \mathcal{T} :: \circ \quad \Delta \vdash t :: \mathcal{T}' :: \circ}{\Delta \vdash P[t] :: \mathcal{T}' :: \circ} \text{ ProcessApp}$$

Table 7.2: Process types and basic terms

$$\frac{\Delta \vdash \Phi \Rightarrow \Psi :: \circ \quad new\ \alpha, \beta}{\Delta \vdash (\alpha \Rightarrow \beta) \triangleright (\Phi \Rightarrow \Psi) \blacktriangleleft \circ}\ \text{Channelnaming}$$

$$\frac{\Delta \vdash t \blacktriangleleft (\alpha \Rightarrow \beta) \triangleright (\Phi \Rightarrow \Psi) \blacktriangleleft \circ}{\Delta \vdash [\alpha \Rightarrow \beta]\ t :: (\Phi \Rightarrow \Psi) :: \circ}\ \text{ChannelAbs}$$

$$\frac{\Delta \vdash t :: \Phi \Rightarrow \Psi :: \circ \quad \Delta \vdash (\alpha \Rightarrow \beta) \triangleright (\Phi \Rightarrow \Psi) \blacktriangleleft \circ}{\Delta \vdash \mathsf{call}\ t\ (\alpha \Rightarrow \beta) \blacktriangleleft (\alpha \Rightarrow \beta) \triangleright \Phi \Rightarrow \Psi \blacktriangleleft \circ}\ \text{Call}$$

Table 7.3: Process types with channels

7.1.4 Message passing: getting and putting

Table 7.4 gives the typing rules for the message passing primitives. When one gets a message of type `T` on a channel, α, which has an output polarity, α must have a protocol $\mathtt{T} \ll \mathtt{P}$ where `T` is the sequential type of the message and `P` is the protocol of the channel after the message has been recieved. As it is an output polarity channel, receiving messages is seen from the perspective of the process itself, i.e. the process is getting a message on channel α. On the other hand, if the channel, α, has input polarity, then that channel must have protocol $\mathtt{T} \gg \mathtt{P}$. As it is an input polarity channel, receiving messages is seen from the perspective of the external world, i.e. the external world (any other process) is putting a message on channel α to be recieved by the process. Similarly when one outputs a message on a channel which has an output polarity, then that channel must have a protocol $\mathtt{T} \gg \mathtt{P}$ where `T` is the sequential type of the message which is being passed and `P` is the protocol of the channel after the message has been sent. On the other hand, if the channel has an input polarity, then the channel must have protocol $\mathtt{T} \ll \mathtt{P}$.

$$\frac{\Delta, x : T : \bullet \vdash P \blacktriangleleft \Phi, \alpha \triangleright T', \Phi' \Rightarrow \Psi \blacktriangleleft \circ}{\Delta \vdash \mathsf{get}\ x\ \mathsf{on}\ \alpha\ ;\ P \blacktriangleleft \Phi, \alpha \triangleright T \gg T', \Phi' \Rightarrow \Psi \blacktriangleleft \circ}\ \text{getL}$$

$$\frac{\Delta, x : T : \bullet \vdash P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright T', \Psi' \blacktriangleleft \circ}{\Delta \vdash \mathsf{get}\ x\ \mathsf{on}\ \alpha\ ;\ P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright T \ll T', \Psi' \blacktriangleleft \circ}\ \text{getR}$$

$$\frac{\Delta \vdash t : T : \bullet \quad \Delta \vdash P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright T', \Psi' \blacktriangleleft \circ}{\Delta \vdash \mathsf{put}\ t\ \mathsf{on}\ \alpha\ ;\ P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright T \gg T', \Psi' \blacktriangleleft \circ}\ \text{putR}$$

$$\frac{\Delta \vdash t : T : \bullet \quad \Delta \vdash P \blacktriangleleft \Phi, \alpha \triangleright T', \Phi' \Rightarrow \Psi \blacktriangleleft \circ}{\Delta \vdash \mathsf{put}\ t\ \mathsf{on}\ \alpha\ ;\ P \blacktriangleleft \Phi, \alpha \triangleright T \ll T', \Phi' \Rightarrow \Psi \blacktriangleleft \circ}\ \text{putL}$$

Table 7.4: Message passing: getting and putting

7.1.5 Closing and ending channels

The rules in Table 7.5 tells how to close and end channels: these rules allow a process to be terminated successfully. The $\top$ and $\bot$ protocols allow communication to be finished on a channel and the channel to be removed. However, removing channels, in this manner must be done in a manner that depends on the channel polarity. One can close a channel when other channels are present. In particular, one can close a channel when it has protocol $\bot$ and an output polarity (`closeR` rule), or one can close a channel when it has protocol $\top$ and an input polarity (`closeL` rule). Provided all other channels have been closed one can end this (last channel) provided either it has protocol $\top$ and an output polarity (`endR` rule) or protocol $\bot$ and an input polarity (`endL` rule). This terminates a process.

Example 7.1.1. Consider the follwing example which shows how to type check the message passing, `close` and `end` constructs. The type checked process receives a message on its input channel, α_1 and send the message along its output channel, α_3. Then the communication on the output polarity channel is ended provided that the input polarity channel is closed.

$$\frac{\Delta \vdash P :: \Phi \Rightarrow \Psi \blacktriangleleft \circ}{\Delta \vdash \mathsf{close}\ \alpha; P \blacktriangleleft \Phi, \alpha \triangleright \top \Rightarrow \Psi \blacktriangleleft \circ}\ \text{closeL}$$

$$\frac{\Delta \vdash P \blacktriangleleft \Phi \Rightarrow \Psi \blacktriangleleft \circ}{\Delta \vdash \mathsf{close}\ \alpha; P \blacktriangleleft \Phi \Rightarrow \alpha \triangleright \bot, \Psi \blacktriangleleft \circ}\ \text{closeR}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash \mathsf{end}\ \alpha \blacktriangleleft \alpha \triangleright \bot \Rightarrow \quad \blacktriangleleft \circ}\ \text{endL}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash \mathsf{end}\ \alpha \blacktriangleleft \quad \Rightarrow \alpha \triangleright \top \blacktriangleleft \circ}\ \text{endR}$$

Table 7.5: Closing and ending channels

$$\dfrac{\dfrac{}{a : A : \bullet \vdash a : A : \bullet}\ \text{TermProj} \qquad \dfrac{\dfrac{a : A : \bullet \quad \mathsf{context}}{a : A : \bullet \vdash \mathtt{end}\ \alpha_3 \blacktriangleleft \quad \Rightarrow \alpha_3 \triangleright \top \blacktriangleleft \circ}\ \text{endR}}{a : A : \bullet \vdash \begin{array}{l}\mathtt{close}\ \alpha_1;\\ \mathtt{end}\ \alpha_3\end{array} \blacktriangleleft \alpha_1 \triangleright \top \Rightarrow \alpha_3 \triangleright \top \blacktriangleleft \circ}\ \text{closeR}}{\vdash \begin{array}{l}\mathtt{put}\ a\ \mathtt{on}\ \alpha_3;\\ \mathtt{close}\ \alpha_1;\\ \mathtt{end}\ \alpha_3\end{array} \quad a : A : \bullet \vdash \begin{array}{l}\mathtt{put}\ a\ \mathtt{on}\ \alpha_3;\\ \mathtt{close}\ \alpha_1;\\ \mathtt{end}\ \alpha_3\end{array} \blacktriangleleft \alpha_1 \triangleright \top \Rightarrow \alpha_3 \triangleright (A \ll \top) \blacktriangleleft \circ}\ \text{putR}$$

$$\dfrac{a : A : \bullet \vdash \begin{array}{l}\mathtt{put}\ a\ \mathtt{on}\ \alpha_3;\\ \mathtt{close}\ \alpha_1;\\ \mathtt{end}\ \alpha_3\end{array} \blacktriangleleft \alpha_1 \triangleright \top \Rightarrow \alpha_3 \triangleright (A \ll \top) \blacktriangleleft \circ}{\vdash \begin{array}{l}\mathtt{get}\ a\ \mathtt{on}\ \alpha_1.\\ \mathtt{put}\ a\ \mathtt{on}\ \alpha_3;\\ \mathtt{close}\ \alpha_1;\\ \mathtt{end}\ \alpha_3\end{array} \blacktriangleleft \alpha_1 \triangleright (A \ll \top) \Rightarrow \alpha_3 \triangleright (A \ll \top) \blacktriangleleft \circ}\ \text{getR}$$

$$\frac{\Delta \vdash t :: \mathcal{T} :: \circ \quad \Delta, Q :: \mathcal{T} :: \circ \quad \mathsf{context}}{\Delta, Q :: \mathcal{T} :: \circ := t \quad \mathsf{context}}\ \text{ProcessDef}$$

$$\frac{\Delta, Q :: \mathcal{T} :: \circ := [\alpha \Rightarrow \beta]\ t, \Delta' \vdash t' :: \mathcal{T} :: \circ}{\Delta, \Delta' \vdash t'\ \mathsf{where}\ Q := [\alpha \Rightarrow \beta]\ t :: \mathcal{T} :: \circ}\ \text{Where}_{\text{P}}$$

$$\frac{\Delta, f : T : \alpha := t, \Delta' \vdash t' :: \mathcal{T} :: \circ}{\Delta, \Delta' \vdash t'\ \mathsf{where}\ f := t :: \mathcal{T} :: \circ}\ \text{Where}_{\text{f}}$$

$$\frac{\Delta, Q :: \mathcal{T} :: \circ := t, \Delta' \quad \mathsf{context}}{\Delta, Q :: \mathcal{T} :: \circ := t, \Delta' \vdash Q :: \mathcal{T} :: \circ}\ \text{ProcessUse}$$

Table 7.6: Process definitions and uses

7.1.6 Process definitions and uses

Table 7.6 shows how a process can be defined, and added to the context. The defined processes can be pulled out from the context by using the $\mathsf{Where_P}$ and ProcessUse rules. The $\mathsf{Where_f}$ rule allows any function available in the context to be used under a process (local definition). These rules act similarly to the rules like FunctionDef, Where, FunctionUse discussed earlier for the sequential world.

Example 7.1.2. In order to use the concurrent term in example 7.1.1 as a process, the channel names need to be abstracted from the process type by using ChannelAbs rule. Then one can define the process and put it in the context if there is a process variable P that has the same type as the term.

$$\frac{\begin{array}{c}\vdots\\ \end{array}}{\vdash \begin{array}{l}[\alpha_1 \Rightarrow \alpha_3]\\ \\ \\ \\ \\ \end{array} \begin{array}{l}\texttt{get}\ a\ \texttt{on}\ \alpha_1.\\ \texttt{put}\ a\ \texttt{on}\ \alpha_3;\\ \texttt{close}\ \alpha_1;\\ \texttt{end}\ \alpha_3\end{array} \begin{array}{l}P :: (A \ll \top) \Rightarrow (A \ll \top)\\ \quad :: \circ\ \ \mathsf{ctxt}\end{array} :: (A \ll \top) \Rightarrow (A \ll \top) :: \circ}\ \text{ChlAbs}$$

$$\frac{}{P :: (A \ll \top) \Rightarrow (A \ll \top) :: \circ := [\alpha_1 \Rightarrow \alpha_3] \begin{array}{l}\texttt{get}\ a\ \texttt{on}\ \alpha_1.\\ \texttt{put}\ a\ \texttt{on}\ \alpha_3;\\ \texttt{close}\ \alpha_1;\\ \texttt{end}\ \alpha_3\end{array} \ \mathsf{ctxt}}\ \text{ProDef}$$

Example 7.1.3. To continue the example 7.1.2: we show how `processUse` can be used to pull out a process from the context and the `call` construct shows the important step of attaching of channels to the process:

$$\frac{\dfrac{\begin{array}{c}\vdots\\ \end{array}}{\Delta\ \ \mathsf{ctxt}}}{\Delta \vdash P :: (A \ll \top) \Rightarrow (A \ll \top) :: \circ}\ \text{ProcessUse} \quad \frac{\begin{array}{c}\vdots\\ \end{array}}{\Delta \vdash (\alpha_1 \Rightarrow \alpha_3) \triangleright (A \ll \top) \Rightarrow (A \ll \top) \blacktriangleleft \circ}}{\Delta \vdash \begin{array}{l}\texttt{call}\ P\ (\alpha_1 \Rightarrow \alpha_3)\\ \blacktriangleleft\ (\alpha_1 \Rightarrow \alpha_3) \triangleright (A \ll \top) \Rightarrow (A \ll \top) \blacktriangleleft \circ\end{array}}\ \text{Call}$$

where, Δ contains $P :: (A \ll \top) \Rightarrow (A \ll \top) :: \circ := [\alpha_1 \Rightarrow \alpha_3]$ $\begin{array}{l}\texttt{get}\ a\ \texttt{on}\ \alpha_1.\\ \texttt{put}\ a\ \texttt{on}\ \alpha_3;\\ \texttt{close}\ \alpha_1;\\ \texttt{end}\ \alpha_3\end{array}$

7.1.7 Channel identification and plugging processes together

Four rules are given in table 7.7 which show the typing of channel identification and process composition. The first one, Identity rule, allows one to identify two channels provided they

$$\frac{\Delta \vdash X : \circ \quad new\ \alpha, \beta}{\Delta \vdash (\alpha == \beta) \blacktriangleleft \alpha \triangleright X \Rightarrow \beta \triangleright X \blacktriangleleft \circ}\ \text{Identity}$$

$$\frac{\Delta \vdash X : \circ \quad new\ \alpha, \beta}{\Delta \vdash (\alpha == neg(\beta)) \blacktriangleleft \alpha \triangleright neg(X), \beta \triangleright X \Rightarrow \quad \blacktriangleleft \circ}\ \text{negL}$$

$$\frac{\Delta \vdash X : \circ \quad new\ \alpha, \beta}{\Delta \vdash (neg(\alpha) == \beta) \blacktriangleleft \quad \Rightarrow \alpha \triangleright X, \beta \triangleright neg(X) \blacktriangleleft \circ}\ \text{negR}$$

$$\frac{\Delta \vdash \begin{array}{l} P\ (\beta_3 \Rightarrow \alpha_1, \alpha, \alpha_2) \\ \blacktriangleleft \beta_3 \triangleright \Phi \Rightarrow \alpha_1 \triangleright \Psi, \alpha \triangleright T, \alpha_2 \triangleright \Psi' \blacktriangleleft \circ \end{array} \qquad \Delta' \vdash \begin{array}{l} Q\ (\beta_1, \beta, \beta_2 \Rightarrow \alpha_3) \\ \blacktriangleleft \beta_1 \triangleright \Phi', \beta \triangleright T, \beta_2 \triangleright \Phi'' \Rightarrow \alpha_3 \triangleright \Psi'' \blacktriangleleft \circ \end{array}}{\Delta, \Delta' \vdash \begin{array}{l} \mathsf{plug\ on}\ (\alpha == \beta) \\ P\ (\beta_3 \Rightarrow \alpha_1, \alpha, \alpha_2) \\ \mathsf{to} \\ Q\ (\beta_1, \beta, \beta_2 \Rightarrow \alpha_3) \end{array} \blacktriangleleft \begin{array}{l} \beta_1 \triangleright \Phi', \beta_3 \triangleright \Phi, \beta_2 \triangleright \Phi^2 \Rightarrow \\ \alpha_1 \triangleright \Psi, \alpha_3 \triangleright \Psi'', \alpha_2 \triangleright \Psi' \end{array} \blacktriangleleft \circ}\ \text{plug}$$

Table 7.7: Channel identification, negation and process composition-plugging

have the same type and opposite polarity. The latter two rules allows one to identify two channels provided they have the same type and same polarity. The last rule in the table allows two processes to be "plugged" together. When plugging two processes together by identifying channels, the types of the channels which are being identified have to be the same.

Example 7.1.4. The following example shows how the plug rule can be used to compose two processes P and Q:

$$\frac{\vdots \qquad\qquad \vdots}{\begin{array}{ll}\Delta \vdash & \texttt{call}\ P\ (\alpha_1 \Rightarrow \alpha_3) \\ & \blacktriangleleft\ (\alpha_1 \Rightarrow \alpha_3) \triangleright (T_1 \Rightarrow T) \blacktriangleleft \circ\end{array} \qquad \begin{array}{ll}\Delta' \vdash & \texttt{call}\ Q\ (\alpha_2 \Rightarrow \alpha_4) \\ & \blacktriangleleft\ (\alpha_2 \Rightarrow \alpha_4) \triangleright (T \Rightarrow T_2) \blacktriangleleft \circ\end{array}}{\begin{array}{ll}\Delta, \Delta' \vdash & \texttt{plug on}\ (\alpha_3 == \alpha_2) \\ & \quad \texttt{call}\ P\ (\alpha_1 \Rightarrow \alpha_3) \\ & \quad \texttt{to} \\ & \quad \texttt{call}\ Q\ (\alpha_2 \Rightarrow \alpha_4)\end{array} \quad \blacktriangleleft\ (\alpha_1 \Rightarrow \alpha_4) \triangleright (T_1 \Rightarrow T_2) \blacktriangleleft \circ}\ \text{Plug}$$

7.1.8 Splitting and forking

The typing rules for splitting and forking channels are given in 7.8. To split a channel, which has an input polarity, it must have protocol, $T \otimes T'$, the "tensor" of T and T'. On the other hand, to split a channel, which has an output polarity, it must have protocol, $T \oplus T'$, the "cotensor" of T and T'. To fork a channel, which has an output polarity, it must have protocol, $T \otimes T'$ and on the other hand to fork a channel, which has an input polarity, it must have protocol, $T \oplus T'$. Splitting introduces new channel names while forking must divide remaining channels between the processes.

7.1.9 Type abstraction over a process

The typing rules related to abstraction over a process are given in 7.9. The first rule allows a process to take a sequential type as an argument and the PTypeAbs rule allows a process to take a protocol as its argument.

Example 7.1.5. Consider another example in which we illustrate how to bundle channels together using $\oplus$ and also the abstraction of type variables over process. This example shows how we can bundle two output polarity channels into one channel while different messages are received on those channels followed by ending the communication.

$$\frac{\Delta \vdash P \blacktriangleleft \Phi, \alpha_1 \triangleright T, \alpha_2 \triangleright T', \Phi' \Rightarrow \Psi \blacktriangleleft \circ \qquad new\ \alpha}{\Delta \vdash split\ \alpha\ into\ (\alpha_1, \alpha_2)\ in\ P \blacktriangleleft \Phi, \alpha \triangleright T \otimes T', \Phi' \Rightarrow \Psi \blacktriangleleft \circ}\ \text{splitL}$$

$$\frac{\Delta \vdash P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha_1 \triangleright T, \alpha_2 \triangleright T', \Psi' \blacktriangleleft \circ \qquad new\ \alpha}{\Delta \vdash split\ \alpha\ into\ (\alpha_1, \alpha_2)\ in\ P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright T \oplus T', \Psi' \blacktriangleleft \circ}\ \text{splitR}$$

$$\frac{\Delta \vdash P_1 \blacktriangleleft \Phi, \alpha_1 \triangleright T : \circ \Rightarrow \Psi_{\beta_1} \blacktriangleleft \circ \quad \Delta' \vdash P_2 \blacktriangleleft \Phi', \alpha_2 \triangleright T' \Rightarrow \Psi'_{\beta_2} \blacktriangleleft \circ \qquad new\ \alpha}{\begin{array}{l} fork\ \alpha\ as \\ \Delta, \Delta' \vdash \quad \alpha_1\ with\ \beta_1 \to P_1 \\ \qquad\quad \alpha_2\ with\ \beta_2 \to P_2 \end{array} \blacktriangleleft \Phi, \alpha \triangleright T \oplus T', \Phi' \Rightarrow \Psi \blacktriangleleft \circ}\ \text{forkL}$$

$$\frac{\Delta \vdash P_1 \blacktriangleleft \Phi_{\beta_1} \Rightarrow \alpha_1 \triangleright T, \Psi \blacktriangleleft \circ \quad \Delta' \vdash P_2 \blacktriangleleft \Phi'_{\beta_2} \Rightarrow \alpha_2 \triangleright T', \Psi' \blacktriangleleft \circ \qquad new\ \alpha}{\begin{array}{l} fork\ \alpha\ as \\ \Delta, \Delta' \vdash \quad \alpha_1\ with\ \beta_1 \to P_1 \\ \qquad\quad \alpha_2\ with\ \beta_2 \to P_2 \end{array} \blacktriangleleft \Phi, \Phi' \Rightarrow \Psi, \alpha \triangleright T \otimes T', \Psi' \blacktriangleleft \circ}\ \text{forkR}$$

Table 7.8: Splitting and forking

$$\frac{\Delta, X : \bullet \vdash P :: \mathcal{T} :: \alpha}{\Delta \vdash \Lambda X.P :: \mathcal{T} :: \bullet \to \alpha}\ \text{STypeAbs}$$

$$\frac{\Delta, X : \circ \vdash P :: \mathcal{T} :: \alpha}{\Delta \vdash \Pi X.P :: \mathcal{T} :: \circ \to \alpha}\ \text{PTypeAbs}$$

Table 7.9: Type abstraction over a process

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A:\bullet, B:\bullet, a:A:\bullet, b:B:\bullet \quad \mathsf{context}}\ \text{endR}}{A:\bullet, B:\bullet, a:A:\bullet, b:B:\bullet \vdash \texttt{end}\ \beta_2 \ \blacktriangleleft \ \Rightarrow \beta_2 \triangleright \top \blacktriangleleft \circ}}{A:\bullet, B:\bullet, a:A:\bullet, b:B:\bullet \vdash \texttt{close}\ \beta_1;\ \texttt{end}\ \beta_2 \ \blacktriangleleft \ \Rightarrow \beta_1 \triangleright \bot, \beta_2 \triangleright \top \blacktriangleleft \circ}\ \text{closeR}}{A:\bullet, B:\bullet, a:A:\bullet \vdash \begin{array}{l}\texttt{get}\ b\ \texttt{on}\ \beta_2;\\ \texttt{close}\ \beta_1;\ \texttt{end}\ \beta_2\end{array} \ \blacktriangleleft \ \Rightarrow \beta_1 \triangleright \bot, \beta_2 \triangleright (B \ll \top) \blacktriangleleft \circ}\ \text{getR}}{A:\bullet, B:\bullet \vdash \begin{array}{l}\texttt{get}\ a\ \texttt{on}\ \beta_1,\ \texttt{get}\ b\ \texttt{on}\ \beta_2;\\ \texttt{close}\ \beta_1;\ \texttt{end}\ \beta_2\end{array} \ \blacktriangleleft \ \Rightarrow \beta_1 \triangleright (A \ll \bot), \beta_2 \triangleright (B \ll \top) \blacktriangleleft \circ}\ \text{getR}}{A:\bullet, B:\bullet \vdash \begin{array}{l}\texttt{split}\ \beta\ \texttt{into}\ (\beta_1, \beta_2)\ \texttt{in}\\ \texttt{get}\ a\ \texttt{on}\ \beta_1;\ \texttt{get}\ b\ \texttt{on}\ \beta_2;\\ \texttt{close}\ \beta_1;\ \texttt{end}\ \beta_2\end{array} \ \blacktriangleleft \ \Rightarrow \beta \triangleright (A \ll \bot) \oplus (B \ll \top) \blacktriangleleft \circ}\ \text{splitR}}{A:\bullet \vdash \begin{array}{l}\texttt{split}\ \beta\ \texttt{into}\ (\beta_1, \beta_2)\ \texttt{in}\\ \texttt{get}\ a\ \texttt{on}\ \beta_1;\ \texttt{get}\ b\ \texttt{on}\ \beta_2;\\ \texttt{close}\ \beta_1;\ \texttt{end}\ \beta_2\end{array} \ \blacktriangleleft \Lambda B. \ \Rightarrow \beta \triangleright (A \ll \bot) \oplus (B \ll \top) \blacktriangleleft \bullet \to \circ}\ \text{STypeAbs}}{\begin{array}{l}\texttt{split}\ \beta\ \texttt{into}\ (\beta_1, \beta_2)\ \texttt{in}\\ \texttt{get}\ a\ \texttt{on}\ \beta_1;\ \texttt{get}\ b\ \texttt{on}\ \beta_2;\\ \texttt{close}\ \beta_1;\ \texttt{end}\ \beta_2\end{array} \ \blacktriangleleft \Lambda A B. \ \Rightarrow \beta \triangleright (A \ll \bot) \oplus (B \ll \top) \blacktriangleleft \bullet \to \bullet \to \circ}\ \text{STypeAbs}$$

7.2 User defined protocols in the concurrent world

In the concurrent world user defined protocols are the analogue of datatypes in the sequential world. They are introduced using the keywords "protocol" or "coprotocol" which correspond, in the concurrent world, to the "data" and "codata" keywords of the sequential world. Protocols are distinguished from coprotocols as they must be "driven" by a channel with input polarity; coprotocols are driven by a channel with an output polarity. This section introduces the typing rules for these user defined protocols.

7.2.1 Protocols

In MPL protocols are defined in much the same way as datatypes:

$$\frac{\{\Delta, \widetilde{X} : \bullet, \widetilde{\$Y} : \circ, \widetilde{\$C} : \circ \vdash F_{ji}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_j : \circ\}_{i=1\cdots k\ and\ j=1\cdots m}}{\begin{array}{l}\mathsf{protocol}\ P_1\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_1 = \{\#c_{1i} : F_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k} \\ \Delta, \quad \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{context} \\ \mathsf{and}\ P_m\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_n = \{\#c_{mi} : F_{mi}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_m\}_{i=1,\cdots,k}\end{array}}\ \text{ProtocolIntro}$$

In the above protocol definition, $P_1, \cdots, P_m$ are the names of the introduced protocols (i.e. the protocol constructors), the $\#c_{1i} \cdots \#c_{mi}$ are process constructors. The fold for these datatypes is called a "drive" to distinguish it from fold in the sequential world. One must drive a program using a user defined protocol on an input polarity channel carrying that protocol.

Example 7.2.1.

(1) Consider the following example where a protocol is introduced into the context:

$$\frac{\dfrac{\dfrac{\vdots}{\Delta\ \ \mathsf{context}}}{\Delta \vdash \mathbb{N} : \bullet}\ \text{TypeCons} \quad \dfrac{\dfrac{\vdots}{\Delta\ \ \mathsf{context}}}{\Delta \vdash C_1 : \circ}\ \text{TypeProj}}{\Delta \vdash \mathbb{N} \gg C_1 : \circ}\ \text{TypePut} \quad \dfrac{\dfrac{\dfrac{\vdots}{\Delta\ \ \mathsf{context}}}{\Delta \vdash \mathbb{N} : \bullet}\ \text{TypeCons} \quad \dfrac{\dfrac{\vdots}{\Delta\ \ \mathsf{context}}}{\Delta \vdash C_1 : \circ}\ \text{TypeProj}}{\Delta \vdash \mathbb{N} \ll \bot : \circ}\ \text{TypeGet}$$

$$\frac{}{\begin{array}{l}\mathsf{protocol}\ \texttt{Ping} \Rightarrow C_1 = \\ \Delta', \quad \#\texttt{Ping} :: \mathbb{N} \gg C_1 \Rightarrow C_1 \quad \mathsf{context} \\ \qquad\quad \#\texttt{Pong} :: \mathbb{N} \ll \bot \Rightarrow C_1\end{array}}\ \text{ProtocolIntro}$$

Where, Δ' contains $\begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathsf{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\quad \mathsf{Succ} : \mathsf{C} \to \mathsf{C}\end{array}$ and Δ contains $\Delta', C_1 : \circ$

(2) Consider the following example where a mutual recursive protocol is introduced into the context:

$$\dfrac{\dfrac{\vdots}{\Delta \vdash A \gg D : \circ}\ \text{TypePut} \quad \dfrac{\vdots}{\Delta \vdash B \ll C : \circ}\ \text{TypeGet}}{\begin{array}{l}\mathsf{protocol}\ \texttt{Talk(AB)} \Rightarrow C = \\ \qquad \#\texttt{response} :: A \gg D \Rightarrow C \qquad \mathsf{context} \\ \mathsf{and}\quad \texttt{Respond(A B)} \Rightarrow D = \\ \qquad \texttt{\#listen} :: B \ll C \Rightarrow D\end{array}}\ \text{ProtocolIntro}$$

Where, $\Delta = C : \circ, D : \circ, A : \bullet, B : \bullet$

7.2.1.1 Typing process constructors

The following rule is the typing rule for process constructors:

$$\dfrac{\Delta \quad \mathsf{context}}{\Delta \vdash \#c_{ni} : \Lambda \widetilde{X}.\Pi \widetilde{\$Y}.F_{ni}\ \widetilde{X}\ \widetilde{\$Y}(P_i\ \widetilde{X}\ \widetilde{\$Y}) : \bullet \to \cdots \to \circ \cdots \to \circ}\ \text{PCons}$$

where, Δ contains

$$\begin{array}{l}\mathsf{protocol}\ P_1\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_1 = \{\#c_{1i} : F_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k} \\ \qquad\qquad \vdots \\ \mathsf{and}\ P_m\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_n = \{\#c_{mi} : F_{mi}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_m\}_{i=1,\cdots,k}\end{array}$$

Example 7.2.2.

Consider the following example where #`response` constructor of the protocol `Talk (A B)` is pulled out from the context:

$$\dfrac{\dfrac{\vdots}{\Delta \quad \mathsf{context}}}{\Delta \vdash \#\texttt{response} : \Lambda A\, B.(A \gg (\texttt{Respond A B})) \to (\texttt{Talk A B}) : \bullet \to \bullet \to \circ}\ \text{PCons}$$

where, Δ contains

$$\begin{array}{l}\mathsf{protocol}\ \texttt{Talk(A B)} \Rightarrow C = \\ \qquad \#\texttt{response} :: A \gg D \Rightarrow C \\ \mathsf{and}\quad \texttt{Respond(A B)} \Rightarrow D = \\ \qquad \texttt{\#listen} :: B \ll C \Rightarrow D\end{array}$$

7.2.1.2 Typing protocol constructors

The following rule shows how to type a protocol constructor:

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash P_i : \bullet \to \cdots \to \circ \cdots \to \circ} \text{ PTypeCons}$$

where, Δ contains

$$\begin{array}{l} \mathsf{protocol}\ P_1\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_1 = \{\#c_{1i} : F_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k} \\ \qquad\qquad \vdots \\ \mathsf{and}\ P_m\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_n = \{\#c_{mi} : F_{mi}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_m\}_{i=1,\cdots,k} \end{array}$$

Example 7.2.3.

In the following example, a protocol constructor is pulled out from the context:

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash \texttt{Talk} : \bullet \to \bullet \to \circ} \text{ PTypeCons}$$

where, Δ contains

$$\begin{array}{l} \mathsf{protocol}\ \texttt{Talk(A B)} \Rightarrow C = \\ \qquad\qquad \#\texttt{response} :: A \gg D \Rightarrow C \\ \mathsf{and}\quad \texttt{Respond(A B)} \Rightarrow D = \\ \qquad\qquad \#\texttt{listen} :: B \ll C \Rightarrow D \end{array}$$

7.2.1.3 Process constructors on a channel

Consider consChannel rule:

$$\frac{\Delta \vdash \#c : F\ \widetilde{X}\ \widetilde{\$Y}(P\ \widetilde{X}\ \widetilde{\$Y}) : \circ \qquad \begin{array}{l} \Delta \vdash P \blacktriangleleft \Phi \Rightarrow \Psi, \\ \qquad \alpha \triangleright F\ \widetilde{X}\ \widetilde{\$Y}(P\ \widetilde{X}\ \widetilde{\$Y}), \Psi' \blacktriangleleft \circ \end{array}}{\Delta \vdash \#c\ on\ \alpha\ ;\ P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright P\ \widetilde{X}\ \widetilde{\$Y}, \Psi' \blacktriangleleft \circ} \text{ consChannel}$$

In this rule, $\#c$ is a process constructor and the rule shows how to type the application of constructor to a channel.

All the rules related to user defined protocols are collected in the Table 7.10.

$$\frac{\{\Delta, \widetilde{X} : \bullet, \widetilde{\$Y} : \circ, \widetilde{\$C} : \circ \vdash F_{ji}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_j : \circ\}_{i=1\cdots k\ and\ j=1\cdots m}}{\begin{array}{l}\Delta, \quad \begin{array}{l}\mathsf{protocol}\ P_1\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_1 = \{\#c_{1i} : F_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k} \\ \vdots \\ \mathsf{and}\ P_m\ \widetilde{X}\ \widetilde{\$Y} \Rightarrow \$C_n = \{\#c_{mi} : F_{mi}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_m\}_{i=1,\cdots,k}\end{array} \quad \mathsf{context}\end{array}}\ \text{ProtocolIntro}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash \#c_{ni} : \Lambda\widetilde{X}.\Pi\widetilde{\$Y}.F_{ni}\ \widetilde{X}\ \widetilde{\$Y}(P_i\ \widetilde{X}\ \widetilde{\$Y}) : \bullet \to \cdots \to \circ \cdots \to \circ}\ \text{PCons}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash P_i : \bullet \to \cdots \to \circ \cdots \to \circ}\ \text{PTypeCons}$$

$$\frac{\Delta \vdash \#c : F\ \widetilde{X}\ \widetilde{\$Y}(P\ \widetilde{X}\ \widetilde{\$Y}) : \circ \qquad \begin{array}{l}\Delta \vdash P \blacktriangleleft \Phi \Rightarrow \Psi, \\ \qquad \alpha \triangleright F\ \widetilde{X}\ \widetilde{\$Y}(P\ \widetilde{X}\ \widetilde{\$Y}), \Psi' \blacktriangleleft \circ\end{array}}{\Delta \vdash \#c\ on\ \alpha\ ;\ P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright P\ \widetilde{X}\ \widetilde{\$Y}, \Psi' \blacktriangleleft \circ}\ \text{consChannel}$$

Table 7.10: Rules related to protocols

7.2.2 Coprotocols

Here we introduce the rules for coprotocols. Like protocols, coprotocol definitions can be added to the context. A coprotocol definition in the context delivers: type constructors (i.e. the name of the coprotocol with its arguments), coprotocol destructors, match, and the drive combinator. Unlike sequential world, in the concurrent world we do not have different recursive combinators for protocols and coprotocols. The symmetry between input and output polarity processes allows us to have one syntax for the recursive combinator, the drive, in the concurrent world.

The first rule introduces a coprotocol into the context:

$$\frac{\{\Delta, \widetilde{X} : \bullet, \widetilde{\$Y} : \circ, \widetilde{\$C} : \circ \vdash G_{ji}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_j : \circ\}_{i=1\cdots k\ and\ j=1\cdots n}}{\begin{array}{l}\mathsf{coprotocol}\ \$C_1 \Rightarrow E_1\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{1i} : G_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k} \\ \Delta, \quad \vdots \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{context} \\ \mathsf{and}\ \$C_n \Rightarrow E_n\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{ni} : G_{ni}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_n\}_{i=1,\cdots,k}\end{array}}\ \text{CoProtocolIntro}$$

Example 7.2.4.

(1) In the following example a coprotocol is introduced into the context:

$$\dfrac{\dfrac{\dfrac{\vdots}{A : \bullet, B : \bullet, C : \bullet\ \ \mathsf{context}}}{A : \bullet, B : \bullet, C : \bullet \vdash A : \bullet}\ \text{TypeProj} \quad \dfrac{\vdots}{A : \bullet, B : \bullet, C : \bullet \vdash (B \ll C) :: \circ}\ \text{TypePut}}{A : \bullet, B : \bullet, C : \bullet \vdash A \gg (B \ll C) :: \circ}\ \text{TypeGet}}{\begin{array}{l}\mathsf{coprotocol}\ C \Rightarrow \mathtt{Respond(AB)} = \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{context} \\ \qquad \#\mathtt{cresponse} :: C \Rightarrow A \gg (B \ll C)\end{array}}\ \text{CoprotocolIntro}$$

(2) In the following example a mutual recursive coprotocol is introduced into the context:

$$\dfrac{\dfrac{\vdots}{\Delta \vdash B \gg D : \circ}\ \text{TypePut} \quad \dfrac{\vdots}{\Delta \vdash A \ll C : \circ}\ \text{TypeGet}}{\begin{array}{l}\mathsf{coprotocol}\ C \Rightarrow \texttt{CoTalk(AB)} = \\ \qquad \#\texttt{coresponse} :: C \Rightarrow (B \gg D) \\ \mathsf{and}\ \ D \texttt{=>}\ \texttt{CoRespond(AB)} = \\ \qquad \#\texttt{colisten} :: D \Rightarrow (A \ll C)\end{array}\ \begin{array}{l}\text{CoprotocolIntro} \\ \\ \mathsf{context}\end{array}}$$

7.2.2.1 Typing process destructors

The following rule is the typing rule for a process destructors:

$$\dfrac{\Delta \quad \mathsf{context}}{\Delta \vdash \#d_{ni} : \Lambda \widetilde{X}.\Pi \widetilde{\$Y}.G_{ni}\ \widetilde{X}\ \widetilde{\$Y}(E_i\ \widetilde{X}\ \widetilde{\$Y}) : \bullet \to \cdots \to \circ \cdots \to \circ}\ \text{CPDes}$$

where Δ contains $\begin{array}{l}\mathsf{coprotocol}\ \$C_1 \Rightarrow E_1\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{1i} : G_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k} \\ \vdots \\ \mathsf{and}\ \$C_n \Rightarrow E_n\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{ni} : G_{ni}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_n\}_{i=1,\cdots,k}\end{array}$

Example 7.2.5.

In the following example, the process destructor `#cresponse` of coprotocol `Respond (A B)` is pulled out from the context:

$$\dfrac{\dfrac{\vdots}{\Delta \quad \mathsf{context}}}{\Delta \vdash \#\texttt{cresponse} : \Lambda A\, B.(\texttt{RespondAB}) \to A \gg (B \ll (\texttt{Respond A B})) : \bullet \to \bullet \to \circ}\ \text{CPDes}$$

where, Δ contains $\begin{array}{l}\mathsf{coprotocol}\ C \Rightarrow \texttt{Respond(A B)} = \\ \qquad \#\texttt{cresponse} :: C \Rightarrow A \gg (B \ll C)\end{array}$

7.2.2.2 Typing coprotocol constructors

The following rule shows how to type check a coprotocol constructors in a coprotocol definition:

$$\dfrac{\Delta \quad \mathsf{context}}{\Delta \vdash E_i : \bullet \to \cdots \to \circ \cdots \to \circ}\ \text{CPTypeCons}$$

$$\mathsf{coprotocol}\ \$C_1 \Rightarrow E_1\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{1i} : G_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k}$$

where Δ contains $\quad\vdots$

$$\mathsf{and}\ \$C_n \Rightarrow E_n\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{ni} : G_{ni}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_n\}_{i=1,\cdots,k}$$

Example 7.2.6.

The following example shows how to extraxt a coprotocol constructor from the context:

$$\frac{\Delta \qquad \mathsf{context}}{\Delta \vdash \texttt{Respond} : \bullet \to \bullet \to \circ}\ \text{CPTypeCons}$$

where, Δ contains $\quad \mathsf{coprotocol}\ C \Rightarrow \texttt{Respond(A B)} =$

$$\#\texttt{cresponse} :: C \Rightarrow A \gg (B \ll C)$$

7.2.2.3 Process destructors on a channel

We can apply a process destructor on any channel. The following rules shows how to type check destructor on a channel:

$$\frac{\Delta \vdash \#d : G\ \widetilde{X}\ \widetilde{\$Y}(E\ \widetilde{X}\ \widetilde{\$Y}) : \circ \qquad \begin{array}{c}\Delta \vdash P \blacktriangleleft \Phi \Rightarrow \Psi, \\ \alpha \triangleright G\ \widetilde{X}\ \widetilde{\$Y}(E\ \widetilde{X}\ \widetilde{\$Y}), \Psi' \blacktriangleleft \circ\end{array}}{\Delta \vdash \#d\ on\ \alpha\ ;\ P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright E\ \widetilde{X}\ \widetilde{\$Y}, \Psi' \blacktriangleleft \circ}\ \text{desChannel}$$

Example 7.2.7. Consider the following example to explore the CPDes and desChannel step in the type system:

$$\dfrac{\dfrac{\vdots}{\Delta \;\; \mathsf{context}}}{\begin{array}{l}\Delta\vdash\\ \#\texttt{response}:\\ (\texttt{Respond A B})\to A\gg(B\ll(\texttt{Respond A B}))\\ :\circ\end{array}}\ \text{CPDes} \qquad \dfrac{\vdots}{\begin{array}{ll}& \texttt{get}\ a\ \texttt{on}\ \alpha.\\ & \texttt{put}\ a\ \texttt{on}\ \beta;\texttt{get}\ b\ \texttt{on}\ \beta.\\ \Delta\vdash & \texttt{put}\ b\ \texttt{on}\ \alpha.\texttt{close}\ \alpha;\texttt{end}\ \beta\\ & \blacktriangleleft\alpha\triangleright A\gg(B\ll(\texttt{Respond A B}))\Rightarrow\\ & \beta\triangleright A\gg(B\ll(\texttt{Respond A B}))\blacktriangleleft\circ\end{array}}$$

$$\text{desChannel}$$

$$\Delta\vdash \begin{array}{l}\#\texttt{response}\ on\ \alpha;\\ \texttt{get}\ a\ \texttt{on}\ \alpha.\\ \texttt{put}\ a\ \texttt{on}\ \beta;\\ \texttt{get}\ b\ \texttt{on}\ \beta.\\ \texttt{put}\ b\ \texttt{on}\ \alpha.\\ \texttt{close}\ \alpha;\\ \texttt{end}\ \beta\end{array} \quad \blacktriangleleft \begin{array}{l}\alpha\triangleright(Respond\ A\ B)\Rightarrow\\ \beta\triangleright A\gg(B\ll(Respond\ A\ B))\end{array} \quad \blacktriangleleft\circ$$

All the rules related to user defined coprotocols are collected in the Table 7.11.

7.2.3 Typing match

The typing rules for match are given here:

$$\dfrac{\{\Delta\vdash P_i::\Phi,\alpha\triangleright F\ \widetilde{X}\ \widetilde{\$Y}\ (P\ \widetilde{X}\ \widetilde{\$Y}),\Phi'\Rightarrow\Psi::\circ\}_{i=1\cdots n}}{\Delta\vdash \begin{array}{l}\mathsf{match}\ \alpha\ \mathsf{as}\\ \quad\{\#\mathsf{c}_i:P_i\}_{i=1\cdots n}\end{array} ::\Phi,\alpha\triangleright P\ \widetilde{X}\ \widetilde{\$Y},\Phi'\Rightarrow\Psi::\circ}\ \text{matchL}$$

$$\dfrac{\{\Delta\vdash E_i::\Psi\Rightarrow\Phi,\alpha\triangleright G\ \widetilde{X}\ \widetilde{\$Y}\ (E\ \widetilde{X}\ \widetilde{\$Y}),\Phi'::\circ\}_{i=1\cdots n}}{\Delta\vdash \begin{array}{l}\mathsf{match}\ \alpha\ \mathsf{as}\\ \quad\{\#\mathsf{d}_i:E_i\}_{i=1\cdots n}\end{array} ::\Psi\Rightarrow\Phi,\alpha\triangleright E\ \widetilde{X}\ \widetilde{\$Y},\Phi'::\circ}\ \text{matchR}$$

The match is similar to case or record operation.

$$\frac{\{\Delta, \widetilde{X} : \bullet, \widetilde{\$Y} : \circ, \widetilde{\$C} : \circ \vdash G_{ji}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_j : \circ\}_{i=1\cdots k\ and\ j=1\cdots n}}{\begin{array}{l}\mathsf{coprotocol}\ \$C_1 \Rightarrow E_1\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{1i} : G_{1i}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_1\}_{i=1,\cdots,k} \\ \Delta, \quad \vdots \qquad\qquad \mathsf{context} \\ \mathsf{and}\ \$C_n \Rightarrow E_n\ \widetilde{X}\ \widetilde{\$Y} = \{\#d_{ni} : G_{ni}\ \widetilde{X}\ \widetilde{\$Y}\ \$C_n\}_{i=1,\cdots,k}\end{array}}\ \text{CoProtocolIntro}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash \#d_{ni} : \Lambda\widetilde{X}.\Pi\widetilde{\$Y}.G_{ni}\ \widetilde{X}\ \widetilde{\$Y}(E_i\ \widetilde{X}\ \widetilde{\$Y}) : \bullet \to \cdots \to \circ \cdots \to \circ}\ \text{CPDes}$$

$$\frac{\Delta \quad \mathsf{context}}{\Delta \vdash E_i : \bullet \to \cdots \to \circ \cdots \to \circ}\ \text{CPTypeCons}$$

$$\frac{\Delta \vdash \#d : G\ \widetilde{X}\ \widetilde{\$Y}(E\ \widetilde{X}\ \widetilde{\$Y}) : \circ \qquad \begin{array}{l}\Delta \vdash P \blacktriangleleft \Phi \Rightarrow \Psi, \\ \qquad \alpha \triangleright G\ \widetilde{X}\ \widetilde{\$Y}(E\ \widetilde{X}\ \widetilde{\$Y}), \Psi' \blacktriangleleft \circ\end{array}}{\Delta \vdash \#d\ on\ \alpha\ ;\ P \blacktriangleleft \Phi \Rightarrow \Psi, \alpha \triangleright E\ \widetilde{X}\ \widetilde{\$Y}, \Psi' \blacktriangleleft \circ}\ \text{desChannel}$$

Table 7.11: Rules related to coprotocols

Example 7.2.8. In the following example, match is used to build the process which can reflect any message that is received on its inout polarity channel:

```
protocol Reflector (a) => $C =
    #Reflect :: put a (get a bottom) => $C
Mirror :: () Reflector (a) =>
Mirror user => =
    match user as
        #Reflect: get n on user . put n on user ; end user
```

The proof tree for the process body is:

$$\dfrac{\dfrac{}{n : A : \bullet \vdash n : A : \bullet}\ \text{TermProj} \qquad \dfrac{}{n : A : \bullet \vdash \mathsf{end}\ user \quad \blacktriangleleft\ user \triangleright \bot \Rightarrow \blacktriangleleft \circ}\ \text{endL}}{\dfrac{n : A : \bullet \vdash \begin{array}{l}\mathsf{put}\ n\ \mathsf{on}\ user;\\ \mathsf{end}\ user\end{array} \quad \blacktriangleleft\ user \triangleright (A \gg \bot) \Rightarrow \blacktriangleleft \circ}{\dfrac{\begin{array}{l}\mathsf{get}\ n\ \mathsf{on}\ user.\\ \vdash\ \mathsf{put}\ n\ \mathsf{on}\ user;\ \blacktriangleleft\ user \triangleright (A \ll (A \gg \bot)) \Rightarrow \blacktriangleleft \circ\\ \mathsf{end}\ user\end{array}}{\Delta \vdash \begin{array}{l}\mathsf{match}\ user\ \mathsf{with}\\ \quad \#\mathtt{Reflect} : \mathsf{get}\ n\ \mathsf{on}\ user.\\ \qquad\qquad \mathsf{put}\ n\ \mathsf{on}\ user;\\ \qquad\qquad \mathsf{end}\ user\end{array}}\ \text{matchL}}\ \text{getR}}\ \text{putR}$$

7.2.4 The `drive` operation

While in the sequential world, MPL has fold for inductive datatypes and unfold for coinductive datatypes, the symmetry in the concurrent world gives rise to only one operation, `drive` for both folds and unfolds. Here is the rule for the drive combinator:

$$\frac{\left\{\begin{array}{l}\Delta, \widetilde{X} : \bullet, \widetilde{V} : \circ, \widetilde{\mathcal{Y}} : \circ, Q_i :: \Phi, \mathcal{Y}_i, \Phi' \Rightarrow \Psi :: \circ, \widetilde{x} : \widetilde{T'} : \bullet \vdash \\ t_{ji} \blacktriangleleft \Phi, \alpha \triangleright ((F_{ji}\ \widetilde{T}\ \widetilde{\mathcal{T}})\ \widetilde{\mathcal{Y}}), \Phi' \Rightarrow \Psi \blacktriangleleft \circ\end{array}\right\}_i}{\text{drive}}\ \text{drive}$$

$$\Delta, \begin{array}{l}\Pi\widetilde{V}.\Lambda\widetilde{X}.\lambda\widetilde{x}.Q_1 :: \widetilde{T'} \to [\widetilde{\alpha_1} \Rightarrow \widetilde{\beta_1}]\ \Phi_1, \alpha \triangleright P_1\ \widetilde{T}\ \widetilde{\mathcal{T}}, \Phi_1' \Rightarrow \Psi_1 :: \circ \\ \qquad := [\widetilde{\alpha_1} \Rightarrow \widetilde{\beta_1}]\ \mathsf{on}\ \gamma_1\ \{\#c_{1i} : t_{1i}\}_i \\ \vdots \\ \Pi\widetilde{V}.\Lambda\widetilde{X}.\lambda\widetilde{x}.Q_n :: \widetilde{T'} \to\ \widetilde{\mathcal{T}}, \Phi_n' \Rightarrow \Psi_n :: \circ \\ \qquad := [\widetilde{\alpha_n} \Rightarrow \widetilde{\beta_n}]\ \mathsf{on}\ \gamma_n\ \{\#c_{ni} : t_{ni}\}_i\end{array}\ \text{context}$$

Example 7.2.9. In the following example, `drive` is used to add numbers and output the state of the counter on demand:

$$\frac{\begin{array}{ll}\pi_1 & \pi_2 \\ \hline \Delta, \Delta' \vdash \mathsf{get}\ i\ \mathsf{on}\ \alpha; \mathsf{call}\ (Counter(add\ n\ i)(\alpha \Rightarrow) & \Delta, \Delta' \vdash \mathsf{put}\ n\ \mathsf{on}\ \alpha; \mathsf{end}\ \alpha \\ \blacktriangleleft (\alpha \triangleright \mathbb{N} \gg X \Rightarrow) \blacktriangleleft \circ & \blacktriangleleft (\alpha \triangleright \mathbb{N} \ll \bot \Rightarrow) \blacktriangleleft \circ\end{array}}{\text{drive}}\ \text{drive}$$

$$\Delta, \begin{array}{l}\lambda n.Counter :: [\alpha \Rightarrow\ \ ]\ \mathbb{N} \to (\alpha \triangleright \texttt{Ping} \Rightarrow\ \ ) :: \circ := \\ \qquad [\alpha \Rightarrow\ \ ]\ \mathsf{by}\ \alpha \\ \qquad\quad \#\texttt{Ping} : \mathsf{get}\ i\ \mathsf{on}\ \alpha;\ \mathsf{call}\ (Counter(add\ n\ i)(\alpha \Rightarrow)) \\ \qquad\quad \#\texttt{Pong} : \mathsf{put}\ n\ \mathsf{on}\ \alpha; \mathsf{end}\ \alpha\end{array}\ \text{context}$$

$$\Delta = \begin{array}{l}\mathsf{protocol}\ \texttt{Ping} \Rightarrow C_1 = \\ \qquad \#\texttt{Ping} :: \mathbb{N} \gg C_1 \Rightarrow C_1\ , \\ \qquad \#\texttt{Pong} :: \mathbb{N} \ll \bot \Rightarrow C_1\end{array} \quad \begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathsf{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\quad \mathsf{Succ} : \mathsf{C} \to \mathsf{C}\end{array}$$

$\Delta' = X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow\ \ ) :: \circ, n : \mathbb{N} : \bullet$

The completed proof will be given in Appendix C.

Chapter 8

Conclusion and Future Work

This thesis describes a type system for the Message Passing Language (MPL). We began with a detailed description of the language which is based on Cockett and Pastro's message passing logic [**?**] with the addition of concurrent datatypes or protocols. This thesis provides a "proof of concept" by developing examples of programs written in MPL. Some more substantial examples are given to illustrate the expressiveness of MPL.

After the description and examples of MPL, a type system for the sequential world was given and followed by a type system for the concurrent world. These two type systems constitute the type system for MPL which is the basis of type checking and type inference in MPL.

Future work This thesis provides a detailed description of MPL with examples and also provides a complete type system for MPL. However, in this thesis we did not explicitly describe the type inference algorithm which allows one to infer types for any MPL program. The type system given here provides the basis of this type inference system; however, one also requires an annotation of the rules and a generalized unification procedure (which includes universally quantified types).

As has been mentioned, there is still no complete implementation of MPL. There is considerable work still required to complete the implementation. It will be useful to compile the language to an abstract machine and to provide an interpreted environment for the programs in order to fully assess the language.

Appendix A

Grammar for Message Passing Language

A **mpl** program consists of a list of definitions followed by a command to run a process. To run a process one uses the keyword **run** followed by the process name, which must start with upper case letter.

```
MPL -> Defns run name_upper .
Defns -> Defns Defn symb_semi | .
```

A definition is either a sequential definition or a concurrent definition. A sequential definition can be a type definition or a function definition. Similarly, a concurrent definition can be a type definition or a process definition.

```
Defn -> SDefn | CDefn.
SDefn -> STypeDefn | SFunctionDefn .
CDefn -> CTypeDefn | CProcessDefn .
```

A.1 User defined types

Sequential type definitions are either data or codata definitions. A data definition starts with the keyword **data** and can be mutually recursive, in which case it consists of multiple clauses connected by the keyword **and**.

```
STypeDefn -> SData | SCodata .
SData -> data SDataClause SDataClauses .
SDataClauses -> SDataClauses and SDataClause | .
```

Each clause of a data definition is started with the sequential type name which is being declared and can have type variables as arguments, then there is a rightarrow followed by a state variable. The state variable is followed by an equal sign,"=", and a sequence of phrases which are specifications of the constructors for the declared type.

```
SDataClause -> STypeSpec symb_arrowr name_lower symb_eq SDataPhrases .
STypeSpec -> name_upper STDefParams .
STDefParams -> STDefParams name_lower | .
SDataPhrases -> SDataPhrase | SDataPhrases SDataPhrase .
```

A constructor specifications start with a constructor name. Constructor names must start with an upper case letter. The constructor is followed by a colon, and the type specification of the constructor.

```
SDataPhrase -> name_upper symb_colon SType symb_semi .
```

The type specification is not only used for specifying a constructor, it is used whenever the type of all sequential terms is required. A type specification can be either a variable of the data or a sequential type constructor applied to a sequence of types or a tuple. The tuple takes care of products of types.

```
SType -> SArrowedTypes .
SArrowedTypes -> STypePart
               | SArrowedTypes symb_arrowr STypePart .
STypePart -> name_lower | STuple | name_upper STParams .
STParams -> STParams STParam | .
STParam -> name_upper | name_lower | STuple .
STuple -> symb_parenl symb_parenr
```

```
        | symb_parenl STypeList symb_parenr .
STypeList -> SType | STypeList symb_product SType .
```

Codata is similar to data but it starts with the keyword **codata** which is followed by a state variable, a rightarrow, and the codata type constructor with its parameters.

```
SCodata -> codata SCodataClause SCodataClauses .
SCodataClauses -> SCodataClauses and SCodataClause | .
SCodataClause -> name_lower symb_arrowr STypeSpec symb_eq SDataPhrases .
```

A.2 User defined protocols

Concurrent type definitions are either protocol or coprotocol definitions. A protocol definition starts with the keyword **protocol**. A protocol definition can be mutually recursive, in which case it consists of multiple clauses connected by the keyword **and**.

```
CTypeDefn -> Protocol | Coprotocol .
Protocol -> protocol CProtocolClause CProtocolClauses .
CProtocolClauses -> CProtocolClauses and CProtocolClause | .
```

Each clause of a protocol definition is started with a protocol constructor, the name of the protocol, followed by its sequential arguments and concurrent arguments. Following a double rightarrow "=>" to the state variable which is always started with "$". The state variable is followed by "=" and a sequence of clauses which are specifications of the process constructors. Types in the concurrent world, unlike those in the sequential world, may take two kinds of parameters namely sequential types, or other protocols. When defining a parameterized protocol, the parameter list must be split in two, with the list of sequential parameters parenthesized: it can be empty but must be there. After the closing parentheses,

the list of concurrent parameters is supplied (it can be empty). This is a pattern which is repeated whenever parameters must be specified in the concurrent world.

```
CProtocolClause -> CTypeSpec symb_darrowr name_dollared
                   symb_eq CDataPhrases .
CTypeSpec -> name_upper CTDefSParams CTDefParams .
CTDefSParams -> symb_parenl STDefParams symb_parenr | .
CTDefParams -> CTDefParams name_dollared | .
CDataPhrases -> CDataPhrase | CDataPhrases CDataPhrase .
```

Process constructor specifications start with a process constructor name. Process constructor names always start with an "#". The constructor is followed by "::" and the type specification of the constructor. Like the data clauses in the sequential world, each clause of the protocol (respectively coprotocol) definition consists of a constructor (respectively destructor) followed by zero or more parameters. These parameters may be another named type, or may be one of the types over which the protocol (respectively coprotocol) has itself been parameterized. Additionally, as with products of sequential types, they may be tensors (or cotensors) over multiple concurrent types. This includes the empty tensor $\top$ (or empty cotensor $\bot$). While the elements of a tensor over concurrent types are seperated by "(x)' (as with elements of a product of sequential types), elements of a cotensor are seperated by the "(+)" symbol.

```
CDataPhrase -> name_hashed symb_dcolon CType symb_darrowr
               name_dollared symb_semi.
```

A.3 Concurrent types (or protocols)

The concurrent type itself may be a named protocol, an anonymous (concurrent) type, a tensor or cotensor of types (including the empty $\top$ and $\bot$), or it may be a transaction type. A transaction type consists of a sequential type which is either sent or recieved (indicated by the keywords **put** and **get**, respectively), followed by the rest of the protocol. A named protocol may be parmeterized over other types, either in the sequential or the concurrent world. As with the protocol definition, the parameter list (if present) must be split into two, with the (possibly empty) list of sequential parameters parenthesized. After the closing parentheses, the (possibly empty) list of concurrent parameters is supplied.

```
CType -> name_dollared | CNamed | CMultip | CMsgPut
       | CMsgGet | top | bottom.
CNamed -> name_upper symb_parenl STParams symb_parenr CNameParams .
CNameParams -> CNameParams CNameParam | .
CNameParam -> name_dollared | name_upper | CMultip | CMsgPut
            | CMsgGet | top | bottom .
CMultip -> symb_parenl symb_parenr
         | symb_parenl CMultip1 .
CMultip1 -> CType symb_parenr
          | CType symb_tensor CTensorParts symb_parenr
          | CType symb_par CCotensorParts symb_parenr .
CTensorParts -> CType | CTensorParts symb_tensor CType .
CCotensorParts -> CType | CCotensorParts symb_par CType .
CMsgPut -> put STParam CNameParam .
CMsgGet -> get STParam CNameParam .
```

A coprotocol is similar to a protocol but it is started with the keyword **coprotocol** which is followed by a state variable and a double rightarrow "=>' followed by the coprotocol name.

```
Coprotocol -> coprotocol CCoprotocolClause CCoprotocolClauses .
CCoprotocolClauses -> CCoprotocolClauses and CCoprotocolClause | .
CCoprotocolClause -> name_dollared symb_darrowr CTypeSpec symb_eq
                     CCodataPhrases .
CCodataPhrases -> CCodataPhrase
                | CCodataPhrases CCodataPhrase .
CCodataPhrase -> name_hashed symb_dcolon name_dollared symb_darrowr
                 CType symb_semi .
```

A.4 Sequential function definitions

Function definitions may optionally be preceded by a line that explicitly assigns type to that function followed by the function name which must start with lower case letter. Clearly, the given type must correspond to the inferred type of the function. The function can have a list of parameters which is followed by"=" and a sequential term.

```
SFunctionDefn -> name_lower SFTypeLine SPatternParams symb_eq
                 STerm symb_semi .
SFTypeLine -> symb_colon SType symb_semi name_lower | .
SPatternParams -> SPatternParams SPatternParam | .
SPatternParam -> name_lower | symb_undersc .
```

A.5 Sequential terms

A sequential term can be either a function call or a product or a case operation or a record construct or a constructor or a fold or unfold operaton which is followed by an optional where clause.

```
STerm -> STerm1 WhereClause .
STerm1 -> SCall | SProd | SCase | SRecord | SCons | SFold | SUnfold .
```

A function call started with the name of the function which may take parameters.

```
SCall -> name_lower SFParams .
SFParams -> SFParams SFParam | .
SFParam -> name_lower | name_upper | SProd .
```

The product is a (possibly empty) comma-seperated list of terms, enclosed in parentheses.

```
SProd -> symb_parenl symb_parenr
       | symb_parenl SProdParts symb_parenr .
SProdParts -> STerm1
            | SProdParts symb_product STerm1 .
```

The sequential case is used to extract terms from a product, or may be used to decompose a datatype, selecting from among the constructors of which it is comprised. It is started with the keyword **case** followed by a sequential term, the keyword **of**, and then one or more case patterns.

```
SCase -> case STerm1 of SCasePatterns
       | case STerm1 of SProdPattern symb_semi .
```

The patterns that can be matched against may therefore be a product of names, a constructor, followed by the names of its arguments, or the symbol "_" . This last pattern (if present) matches any term that is not matched to another preceding pattern.

```
SCasePatterns -> SCasePattern SCasePatterns1 symb_semi SElsePattern .
SCasePatterns1 -> SCasePatterns1 symb_semi SCasePattern | .
SCasePattern -> name_upper SPatternParams symb_arrowr STerm1 .
SElsePattern -> symb_undersc symb_arrowr STerm1 symb_semi | .
SProdPattern -> symb_parenl symb_parenr symb_arrowr STerm1
              | symb_parenl SProdPattern1 symb_parenr symb_arrowr STerm1 .
SProdPattern1 -> name_lower | SProdPattern1 symb_product name_lower .
```

The dual of the sequential case is record which is started with the keyword **record**. This is used to construct codata by associating values to the destructors.

```
SRecord -> record SRecordParts symb_semi .
SRecordParts -> SRecordPart
              | SRecordParts symb_semi SRecordPart .
SRecordPart -> name_upper symb_arrowl STerm1 .
```

One may apply either a constructor or a destructor to its arguments:

```
SCons -> name_upper SFParams .
```

A fold over a datatype allows for multiple named circular rules, preceded by the keyword **fold** and seperated by the keyword **and**. Each rule may take multiple parameters, with the parameter on which the recursion will act specified by the keyword **by**.

```
SFold -> fold SFoldPhrases .
```

```
SFoldPhrases -> SFoldPhrase | SFoldPhrases and SFoldPhrase .
SFoldPhrase -> name_lower SPatternParams by name_lower as
               SFoldConstrs symb_semi .
SFoldConstrs -> SFoldConstr
              | SFoldConstrs symb_semi SFoldConstr symb_semi .
SFoldConstr -> name_upper SPatternParams symb_arrowr STerm1 .
```

The dual to the fold is unfold. It has similar syntax as fold but it is started with the keyword **unfold**.

```
SUnfold -> unfold SUnfoldPhrases .
SUnfoldPhrases -> SUnfoldPhrase
                | SUnfoldPhrases and SUnfoldPhrase .
SUnfoldPhrase -> name_lower SPatternParams as SUnfoldDestrs symb_semi .
SUnfoldDestrs -> SUnfoldDestr
               | SUnfoldDestrs symb_semi SUnfoldDestr symb_semi .
SUnfoldDestr -> name_upper symb_arrow STerm1 .
```

The keyword **where** may follow any term, allowing the definition of further local functions:

```
WhereClause -> where SFunctionDefns | .
SFunctionDefns -> SFunctionDefns SFunctionDefn symb_semi | .
```

A.6 Process definitions

Process definition starts with process name which must starts with upper case letter. It is follwed by the process type. In the process definition, arguments are specified in the

following order: the list of names corresponding to sequential term parameters which is enclosed in parentheses, the name corresponding to the process parameter is optional, but if present enclosed in curly braces, the list of names corresponding to input channels which are seperated by commas. This is followed by a double arrow, then the required comma separated list of names corresponding to output channels.

```
CProcessDefn -> name_upper CPTypeLine CPParams symb_eq CTerm symb_semi .
CPParams -> CPatternPParam CPatternVParams CPatternParams | .
CPatternPParam -> symb_bracel name_upper symb_bracer | .
CPatternVParams -> symb_parenl SPatternParams symb_parenr | .
CPatternParams -> CChannels symb_darrowr CChannels .
CChannels -> name_lower CChannels1 | .
CChannels1 -> CChannels1 symb_comma name_lower | .
```

Process is always preceded by a line that explicitly assigns a type to that process. A process type line mirrors the structure of the argument list, where instead of a name for each argument, the type of the intended argument is given.

```
CPTypeLine -> CCType symb_dcolon CPType symb_semi name_upper.
CCType -> symb_bracel CPType symb_bracer | .
CPType -> CCType symb_parenl CPSTypes symb_parenr CPTParams
          symb_darrowr CPTParams .
CPSTypes -> SType STypes | .
STypes -> STypes symb_comma SType | .
CPTParams -> CType CPTParams1 | .
CPTParams1 -> CPTParams1 symb_comma CType | .
```

A.7 Concurrent terms

A concurrent term can be formed in one of the 13 ways each of them described seperately below:

```
CTerm -> CTerm1 WhereClause .
CTerm1 -> CID | CCall | CSplit | CFork | CClose | CEnd
        | CGet | CPut | CConstr | CMatch | CCase | CPlug | CDrive .
```

This identity term coresponds to channel identity that specifies that an input channel is to be connected directly to an output channel.

```
CID -> name_lower symb_deq name_lower .
```

When calling another process, the ususal structure (i.e. the same as is used when defining a process) is used for specifying arguments. A comma-separated list of sequential terms is enclosed in parentheses. This is followed by an optional process name enclosed in curly braces, then two comma-separated lists of channel names, with the two lists partitioned by a double arrow "=>". If the process being called does not take any arguments, or its arguments are all already in the context, with their intended names (e.g. when recursing inside a circular term), then the argument list mat be omitted entirely.

```
CCall -> call name_upper CCallParams .
CCallParams -> CCallPParam CCallSParams CChannels symb_darrowr CChannels | .
CCallPParam -> symb_bracel name_upper symb_bracer | .
CCallSParams -> symb_parenl symb_parenr
              | symb_parenl STerm1 STerms symb_parenr | .
STerms -> STerms symb_comma STerm1 | .
```

A channel splitting consists of assigning names to channels into which a channel is to be split and it is preceded by a keyword **split**. This is followed by the channel name which is going to be split and a keyword **into**. This is followed by a comma-separated list of channel names, enclosed in parentheses and rightarrow preceeded by a process.

```
CSplit -> split name_lower into symb_parenl CChannels symb_parenr
          symb_arrowr CTerm1 .
```

A channel forking is the dual of channel splitting. This starts with the keyword **fork** which is followed by a channel name to be forked and the keyword **as**. This is followed by forked parts where each consists of the forked channel names with optional assciated channel name and an assigned term which can be called.

```
CFork -> fork name_lower as CForkParts symb_semi .
CForkParts -> CForkPart
            | CForkParts symb_semi CForkPart .
CForkPart -> name_lower with CChannels symb_arrowr CTerm1 .
```

The closing of a channel is started with the keyword **close** which is follwed by the channel name. This is follwed by a concurrent term continuing the process. To end a process one starts with the keyword **end** followed by the channel name.

```
CClose -> close name_lower symb_semi CTerm1 .
CEnd -> end name_lower .
```

Message passing is performed using the get and put operations. The receiving of message is done with a **get** operation. This comes in the following order: the keyword **get**, then the message (sequential term) that is going to be recieved, the keyword **on**, the channel on which the message is going to be recieved. This is followed by a concurrent term in which

the received message is bound. For sending a message the keyword **put**, then the message (a sequential term) that is going to be sent, the keyword **on**, and the channel on which the message is going to be sent. This is followed by a process to be continued after sending the message.

```
CGet -> get name_lower on name_lower symb_bdot CTerm1 .
CPut -> put STerm1 on name_lower symb_semi CTerm1 .
```

Putting a constructor onto a channel to unwrap a user defined protocol is achieved by writing the constructor's and channel's names.

```
CConstr -> name_hashed On name_lower symb_semi CTerm1 .
```

Matching on a channel reveals its constructor, analogus to casing on a sequential term. This starts with the keyword **match** followed by the channel name and the keyword **as**. This is followed by the constructor that one wants to see on the channel and a concurrent term.

```
CMatch -> match name_lower as CMatchParts .
CMatchParts -> CMatchPart CMatchParts1 symb_semi CElsePart .
CMatchParts1 -> CMatchParts1 symb_semi CMatchPart | .
CMatchPart -> name_hashed symb_colon CTerm1 .
CElsePart -> symb_undersc symb_colon CTerm1 symb_semi | .
```

Casing out a term in the context shares an identical syntax to the case sequential term operation, save that each case specifies a concurrent term, not a sequential term.

```
CCase -> case STerm1 of  CCasePatterns
       | case STerm1 of CProdPattern symb_semi .
```

```
CProdPattern -> symb_parenl symb_parenr symb_arrowr CTerm1
              | symb_parenl SProdPattern1 symb_parenr symb_arrowr CTerm1 .
CCasePatterns -> CCasePattern CCasePatterns1 symb_semi CElsePattern .
CCasePatterns1 -> CCasePatterns1 symb_semi CCasePattern | .
CCasePattern -> name_upper SPatternParams symb_arrowr CTerm1 .
CElsePattern -> symb_undersc symb_arrowr CTerm1 symb_semi | .
```

To arrange for two processes to interact on a channel one plugs them together, and each process is described using the same syntax as it is used to call a process. But it starts with the keyword **plug** which is followed by another keyword **on**. This is followed by channel name on which the plugging between two processes are to interact.

```
CPlug -> plug On name_lower name_upper CCallParams to
         name_upper CCallParams .
```

Finally a recursive process is written using a similar syntax to the fold in the sequential world but it starts with the keyword **drive**. One also needs to indicate by which channel the process is going to be driven (whereas in case of **fold** one needs to indicate the sequential term on which the recursion will act).

```
CDrive -> drive CDrivePhrases symb_semi .
CDrivePhrases -> CDrivePhrase
               | CDrivePhrases symb_semi and CDrivePhrase .
CDrivePhrase -> name_upper CPParams by name_lower as CDriveConstrs symb_semi .
CDriveConstrs -> CDriveConstr
               | CDriveConstrs symb_semi CDriveConstr .
CDriveConstr -> name_hashed symb_colon CTerm1 .
```

Appendix B

Layout rules for MPL

In this appendix, the description of the layout rules for MPL are given.

The layout is handled by a preprocessing step between lexing and parsing. The first preprocessing step is to remove comments. The next step is to identify lexographical tokens: each token has a row and column number associated with it. In the next step the layout is handled: the layout causes explicit tokens (layout semicolons), which can be seen as off-side markers, to be added to the places determined by the layout algorithm. In MPL, one is not allowed to write layout insensitive programs. Layout sensitive programs are converted into layout insensitive programs internally by adding semicolons: the layout insensitive programs are then used by the parser.

B.1 What is Off-Side rule?

The idea of the offside rule was described by Peter J. Landin, in [**?**]. He expressed it by: "Any non-whitespace token to the left of the first such token on the previous line is taken to be the start of a new declaration." The main advantage of an offside rule is that it replaces matching parenthesis or explicit separators by matching indentation which is more convenient to a reader and programmer. MPL employs Landin's idea of an offside rule, with some additional of layout rules which are partially motivated by the layout rules of the Haskell Report [**?**].

B.2 Layout Rule

Before going into the detail description of layout rules, we introduce two terms:

Indentation of a token: The indentation of a token is the column number of the first character of that token.

Indentation of a line: The indentation of a line is the column number of the first character of the first token of the line.

By adding semicolons as off-side markers to the tokens, we produce layout insensitive form of the tokens. To do this, a stack of indentations, the **Indentation Context** is maintained starting with empty stack.

The layout program is divided into two parts: building the indentation context and adding offside markers to the token list using offside rules.

B.2.1 Building Indentation Context

The indentation stack is be built gradually from the empty stack, as the layout program goes through the token list, using the following rules:

- When the indentation context is empty and a token is encountered then the indentation of that token is added to the stack.
- If one of the keywords *as, where, in, fold, drive* or *of* keywords or = is encountered, the indentation is set to the indentation of the next token. This means, the indentation of the next token is pushed onto the indentation stack.

B.2.2 Offside Rules

- When the input token list is empty then the output of the layout function will be empty that is no offside marker (semicolon) is needed.

- We will compare the column number of the current token to the head of the indentation stack:

  If the current column number is equal to the head of the stack then we add an offside marker (a semicolon) to the output token list and continue with the rest of the tokens.

  If the current column number is less than the head of the stack then we will add an offside marker to the token list and pop the head off of the indentation stack and continue with the rest of the tokens.

  Otherwise (that is, if the current column number is greater than the head of the stack) we just continue with the rest of the tokens.

Example B.2.1.

Consider the following program (layout sensitive program) written in MPL:

```
1 data Bool -> c = True : c
2                  False : c
3 or : Bool -> Bool -> Bool
4 or x y =
5    case x of
6         True  -> True
7         False -> y
```

One must write MPL programs using layout. But internally the same program with explicit semicolon separators (after applying layout rule and making it layout insensitive by adding offside markers) looks like :

```
1 data Bool -> c = True : c
2                 ;False : c
3 ;;or : Bool -> Bool -> Bool
4 ;or x y =
5    case x of
6        True  -> True
7       ;False -> y
```

We can also write the program equivalently with explicit semicolons in the following manner:

```
1 data Bool -> c = True : c; False : c
2 or : Bool -> Bool -> Bool; or x y =
3        case x of
4             True  -> True; False -> y
```

Example B.2.2. Consider the following layout sensitive program written in MPL:

```
1  data Nat -> c = Zero : c; Succ : c -> c
2  protocol Ping => $C = #Ping :: put Nat $C => $C
3                       #Pong :: get Nat bottom => $C
4  fold add : Nat -> Nat
5       add x y by x =
6               Zero   -> y
7               Succ n -> Succ (add n y)
8  drive
9     PingCounter :: () Ping() =>
10    PingCounter (n) pinger => by pinger =
11        #Ping: get i on pinger . call PingCounter (add n i) pinger =>
12        #Pong: put n on pinger; end pinger
13
14 run PingCounter
```

One must write the program using layout. But internally, the same program with explicit semicolon separators looks like :

```
1  data Nat -> c = Zero : c; Succ : c -> c
2  ;;protocol Ping => $C = #Ping :: put Nat $C => $C
3                         ;#Pong :: get Nat bottom => $C
4  ;;fold add : Nat -> Nat
5        ;add x y by x =
6               Zero   -> y
7              ;Succ n -> Succ (add n y)
8  ;;drive
9     PingCounter :: () Ping() =>
10    ;PingCounter (n) pinger => by pinger =
11        #Ping: get i on pinger . call PingCounter (add n i) pinger =>
12       ;#Pong: put n on pinger; end pinger
13
14 ;;;run PingCounter
```

Appendix C

Extended examples of MPL

Example C.0.3.

The extended proof of example 6.4.7 is given below:

$$\dfrac{\dfrac{\pi_1}{\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet,\ x : \texttt{Nat} : \bullet, y : \texttt{Nat} : \bullet \vdash y : \texttt{Nat} : \bullet} \quad \dfrac{\pi_2}{\begin{array}{l}\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet,\\ x : \texttt{Nat} : \bullet, y : \texttt{Nat} : \bullet, n : X : \bullet\\ \vdash \texttt{Succ}\ (add\ n\ y) : \texttt{Nat} : \bullet\end{array}}}{\begin{array}{l}\textsf{fold}\\ \Delta,\quad \begin{array}{l} add : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}\\ add\ x\ y\ \texttt{by}\ x := \qquad\qquad \textsf{context}\\ \quad \texttt{Zero} \to y\\ \quad \texttt{Succ}\ n \to \texttt{Succ}\ (add\ n\ y)\end{array}\end{array}}\ \text{fold}$$

Here, $\Delta =$ $\begin{array}{l}\textsf{data}\ \texttt{Nat} \to \mathsf{C} = \texttt{Zero} : \mathsf{C}\\ \qquad\qquad\qquad\quad \texttt{Succ} : \mathsf{C} \to \mathsf{C}\end{array}$

$\pi_1 =$

$$\dfrac{\dfrac{\pi_3}{\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet, x : X : \bullet, y : \texttt{Nat} : \bullet\ \textsf{context}}}{\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet, x : X : \bullet, y : \texttt{Nat} : \bullet \vdash y : \texttt{Nat} : \bullet}\ \text{TermProj}$$

$\pi_3 =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_4}{\Delta \text{ context}}}{\Delta, X : \bullet \text{ context}}}{\Delta, X : \bullet \vdash X : \bullet}\ \text{TypeProj} \quad \dfrac{\dfrac{\dfrac{\pi_4}{\Delta \text{ context}}}{\Delta, X : \bullet \text{ context}}}{\Delta, X : \bullet \vdash \texttt{Nat} : \bullet} \quad \dfrac{\dfrac{\pi_4}{\Delta \text{ context}}}{\Delta, X : \bullet \text{ context}}}{\Delta, X : \bullet \vdash \texttt{Nat} \to \texttt{Nat} : \bullet}\ \text{Implication}}{\Delta, X : \bullet \vdash X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet}\ \text{Implication} \quad \textit{new add}$$

$$\dfrac{\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet \text{ context}}{\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet,\ x : X : \bullet \vdash X : \bullet}\ \text{TypeProj} \quad \textit{new } x$$

$$\dfrac{\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet, x : X : \bullet \text{ context}}{\Delta, X : \bullet, add : X \to (\texttt{Nat} \to \texttt{Nat}) : \bullet,\ x : X : \bullet \vdash \texttt{Nat} : \bullet}\ \text{TypeCons} \quad \textit{new } y$$

$$\Delta, X : \bullet,\ add : X \to (\texttt{Nat} \to \texttt{Nat}),\ : \bullet, x : X : \bullet, y : \texttt{Nat} : \bullet \text{ context}\ \text{TypeVarIntro}$$

$\pi_4 =$

$$\dfrac{\dfrac{\pi_5}{C : \bullet \vdash C : \bullet} \quad \dfrac{\dfrac{\pi_5}{C : \bullet \vdash C : \bullet} \quad \dfrac{\pi_5}{C : \bullet \vdash C : \bullet}}{C : \bullet \vdash C \to C : \bullet}\ \text{Imp}}{\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \texttt{Zero} : \mathsf{C} \quad \texttt{Succ} : \mathsf{C} \to \mathsf{C}}\ \text{DataIntro} \quad \mathsf{context}$$

$\pi_5 =$

$$\dfrac{\dfrac{}{\varepsilon \ \text{context}}\ \text{EmpCtxt} \quad \mathsf{new\ C}}{\dfrac{C : \bullet \ \text{context}}{C : \bullet \vdash C : \bullet}\ \text{TypeProj}}\ \text{TypeVIntro}$$

$\pi_2 =$

$$\dfrac{\dfrac{\dfrac{\pi_6}{\Delta, \Delta' \quad \mathsf{context}}}{\Delta, \Delta' \vdash \mathtt{Succ} : \mathtt{Nat} \to \mathtt{Nat} : \bullet}\ \text{Cons} \quad \dfrac{\pi_7}{\Delta, \Delta' \vdash \mathit{add}\ n\ y : \mathtt{Nat} : \bullet}}{\begin{array}{l}\Delta, X : \bullet, \mathit{add} : X \to (\mathtt{Nat} \to \mathtt{Nat}) : \bullet, x : X : \bullet, y : \mathtt{Nat} : \bullet, n : X : \bullet \\ \vdash \mathtt{Succ}\ (\mathit{add}\ n\ y) : \mathtt{Nat} \to \mathtt{Nat} : \bullet\end{array}}\ \text{TermApp}$$

Here, $\Delta' = X : \bullet, \mathit{add} : X \to (\mathtt{Nat} \to \mathtt{Nat}) : \bullet, x : X : \bullet, y : \mathtt{Nat} : \bullet, n : X : \bullet$

$\pi_6 =$

$$\dfrac{\dfrac{\pi_3}{\Delta, X : \bullet, \mathit{add} : X \to (\mathtt{Nat} \to \mathtt{Nat}) : \bullet, x : X : \bullet, y : \mathtt{Nat} : \bullet\ \mathsf{context}}\ \text{TypVar}}{\begin{array}{l}\Delta, X : \bullet, \mathit{add} : X \to (\mathtt{Nat} \to \mathtt{Nat}) : \bullet, \\ x : X : \bullet, y : \mathtt{Nat} : \bullet \vdash X : \bullet\end{array} \quad \mathit{new}\ n}\ \text{TermVarIntr}$$
$$\Delta, \Delta'\ \mathsf{context}$$

$\pi_7 =$

$$\dfrac{\dfrac{\dfrac{\pi_6}{\Delta, \Delta'\ \mathsf{ctxt}}}{\Delta, \Delta' \vdash \mathit{add} : X \to \mathtt{Nat} \to \mathtt{Nat} : \bullet}\ \text{FunUse} \quad \dfrac{\dfrac{\pi_6}{\Delta, \Delta'\ \mathsf{ctxt}}}{\Delta, \Delta' \vdash n : \mathtt{Nat} : \bullet}\ \text{TermPrj}}{\Delta, \Delta' \vdash \mathit{add}\ n : \mathtt{Nat} \to \mathtt{Nat} : \bullet}\ \text{TermApp} \quad \dfrac{\dfrac{\pi_6}{\Delta, \Delta'\ \mathsf{ctxt}}}{\Delta, \Delta' \vdash y : \mathtt{Nat} : \bullet}\ \text{TermPrj}$$
$$\Delta, \Delta' \vdash \mathit{add}\ n\ y : \mathtt{Nat} : \bullet\ \text{TermApp}$$

Example C.0.4.

The extended proof of example 6.4.8 is given below:

$$\dfrac{\dfrac{\pi_1}{\Delta, \Delta', () : 1 : \bullet \vdash \texttt{True} : \mathbb{B}} \quad \dfrac{\pi_2}{\Delta, \Delta', n_1 : X : \bullet \vdash odd(n_1) : \mathbb{B}} \quad \dfrac{\pi_1}{\Delta, \Delta', () : 1 : \bullet \vdash \texttt{True} : \mathbb{B}} \quad \dfrac{\pi_4}{\Delta, \Delta', n_2 : X : \bullet \vdash even(n_2) : \mathbb{B}}}{\Delta, \begin{array}{l} fold \\ \quad even : \mathbb{N} \to \mathbb{B} \\ \quad even\ x\ \texttt{by}\ x\ as \\ \quad\quad \texttt{Zero} \mapsto \texttt{True} \\ \quad\quad \texttt{Succ}\ n_1 \mapsto odd(n_1) \\ \quad odd : \mathbb{N} \to \mathbb{B} \\ \quad odd\ x\ \texttt{by}\ x\ as \\ \quad\quad \texttt{Zero} \mapsto \texttt{False} \\ \quad\quad \texttt{Succ}\ n_2 \mapsto even(n_2) \end{array} \vdash \texttt{True} : \mathbb{B}}\ \text{fold}$$

where $\text{context} = \Delta, \begin{array}{l} fold \\ \quad even : \mathbb{N} \to \mathbb{B} \\ \quad even\ x\ \texttt{by}\ x\ as \\ \quad\quad \texttt{Zero} \mapsto \texttt{True} \\ \quad\quad \texttt{Succ}\ n_1 \mapsto odd(n_1) \\ \quad odd : \mathbb{N} \to \mathbb{B} \\ \quad odd\ x\ \texttt{by}\ x\ as \\ \quad\quad \texttt{Zero} \mapsto \texttt{False} \\ \quad\quad \texttt{Succ}\ n_2 \mapsto even(n_2) \end{array}$

$$\Delta = \begin{array}{l} \texttt{data}\ \mathbb{N} \to \mathsf{C} = \texttt{Zero} : \mathsf{C} \\ \qquad\qquad\quad \texttt{Succ} : \mathsf{C} \to \mathsf{C} \end{array}, \quad \begin{array}{l} \texttt{data}\ \mathbb{B} \to \mathsf{C} = \texttt{True} : \mathsf{C} \\ \qquad\qquad\quad\ \texttt{False} : \mathsf{C} \end{array}$$

$$\Delta' = X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}$$

$$\pi_1 = \dfrac{\dfrac{\vdots}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, () : 1 : \bullet \quad \text{context}}}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, () : 1 : \bullet \vdash \texttt{True} : \mathbb{B}}\ \text{Cons}$$

$\pi_2 =$

$$\dfrac{\dfrac{\vdots}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_1 : X : \bullet \quad \mathsf{context}}\ \text{TermProj} \qquad \dfrac{\vdots}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_1 : X : \bullet \quad \mathsf{context}}\ \text{TermProj}}{\dfrac{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_1 : X : \bullet \vdash n_1 : \mathbb{N} \qquad \Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_1 : X : \bullet \vdash odd : \mathbb{N} \to \mathbb{B}}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_1 : X : \bullet \vdash odd(n_1) : \mathbb{B}}\ \text{Cons}}$$

$\pi_3 =$

$$\dfrac{\dfrac{\pi}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, () : 1 : \bullet \quad \mathsf{context}}}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, () : 1 : \bullet \vdash \texttt{True} : \mathbb{B}}\ \text{Cons}$$

$\pi_4 =$

$$\dfrac{\dfrac{\vdots}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_2 : X : \bullet \quad \mathsf{context}}\ \text{TermProj} \qquad \dfrac{\vdots}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_2 : X : \bullet \quad \mathsf{context}}\ \text{TermProj}}{\dfrac{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_2 : X : \bullet \vdash n : \mathbb{N} \qquad \Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_2 : X : \bullet \vdash even : \mathbb{N} \to \mathbb{B}}{\Delta, X : \bullet, even : X \to \mathbb{B}, odd : X \to \mathbb{B}, n_2 : X : \bullet \vdash even(n_2) : \mathbb{B}}\ \text{Cons}}$$

Example C.0.5.

The extended proof of example 6.4.6 is given below:

$$
\dfrac{\vdots}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet,\ x:X:\bullet, y:Y:\bullet \ \ \textsf{context}}
$$

$$
\dfrac{X:\bullet, Y:\bullet, t:(X\times Y):\bullet,\ x:X:\bullet, y:Y:\bullet \ \ \textsf{context}}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet,\ x:X:\bullet, y:Y:\bullet \vdash y:Y:\bullet}\ \text{TermProj}
\qquad
\dfrac{X:\bullet, Y:\bullet, t:(X\times Y):\bullet,\ x:X:\bullet, y:Y:\bullet \ \ \textsf{context}}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet,\ x:X:\bullet, y:Y:\bullet \vdash x:X:\bullet}\ \text{TermProj}
$$

$$
\dfrac{X:\bullet, Y:\bullet, t:(X\times Y):\bullet,\ x:X:\bullet, y:Y:\bullet \vdash y:Y:\bullet \qquad X:\bullet, Y:\bullet, t:(X\times Y):\bullet,\ x:X:\bullet, y:Y:\bullet \vdash x:X:\bullet}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet, x:X:\bullet, y:Y:\bullet \ \vdash (y,x):(Y\times X):\bullet}\ \text{TermProd}
$$

$$
\dfrac{X:\bullet, Y:\bullet, t:(X\times Y):\bullet, x:X:\bullet, y:Y:\bullet \ \vdash (y,x):(Y\times X):\bullet}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet, (x,y):(X\times Y):\bullet \ \vdash (y,x):(Y\times X):\bullet}\ \text{ProdPatt}
$$

$$
\dfrac{\dfrac{\vdots}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet \ \ \textsf{context}}}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet \ \vdash t:(X\times Y):\bullet}\ \text{TermProj}
$$

$$
\dfrac{X:\bullet, Y:\bullet, t:(X\times Y):\bullet \ \vdash t:(X\times Y):\bullet \qquad X:\bullet, Y:\bullet, t:(X\times Y):\bullet, (x,y):(X\times Y):\bullet \ \vdash (y,x):(Y\times X):\bullet}{X:\bullet, Y:\bullet, t:(X\times Y):\bullet \vdash \begin{array}{c}\texttt{case}\ t\ \texttt{of}\\ (x,y)\to(y,x)\end{array} :(Y\times X):\bullet}\ \text{case}_{\text{p}}
$$

$$
\dfrac{X:\bullet, Y:\bullet, t:(X\times Y):\bullet \vdash \begin{array}{c}\texttt{case}\ t\ \texttt{of}\\ (x,y)\to(y,x)\end{array} :(Y\times X):\bullet}{X:\bullet, Y:\bullet \vdash \begin{array}{c}\lambda t.\texttt{case}\ t\ \texttt{of}\\ (x,y)\to(y,x)\end{array} :(X\times Y)\to(Y\times X):\bullet}\ \lambda-\text{abst}
$$

$$
\dfrac{X:\bullet, Y:\bullet \vdash \begin{array}{c}\lambda t.\texttt{case}\ t\ \texttt{of}\\ (x,y)\to(y,x)\end{array} :(X\times Y)\to(Y\times X):\bullet}{X:\bullet \vdash \begin{array}{c}\Lambda Y.\lambda t.\texttt{case}\ t\ \texttt{of}\\ (x,y)\to(y,x)\end{array} :\Lambda Y.(X\times Y)\to(Y\times X):\bullet\to\bullet}\ \text{TypeAbs}
$$

$$
\dfrac{X:\bullet \vdash \begin{array}{c}\Lambda Y.\lambda t.\texttt{case}\ t\ \texttt{of}\\ (x,y)\to(y,x)\end{array} :\Lambda Y.(X\times Y)\to(Y\times X):\bullet\to\bullet}{\vdash \begin{array}{c}\Lambda X,Y.\lambda t.\texttt{case}\ t\ \texttt{of}\\ (x,y)\to(y,x)\end{array} :\Lambda X,Y.(X\times Y)\to(Y\times X):\bullet\to(\bullet\to\bullet)}\ \text{TypeAbs}
$$

Example C.0.6.

The extended proof of example 6.4.5 is given below:

$$\dfrac{\dfrac{\pi_1}{\Delta, a : \texttt{Nat} : \bullet, () : 1 : \bullet \vdash \texttt{Zero} : \texttt{Nat} : \bullet} \quad \dfrac{\pi_2}{\Delta, a : \texttt{Nat} : \bullet, n : \texttt{Nat} : \bullet \vdash n : \texttt{Nat} : \bullet}}{\Delta, a : \texttt{Nat} : \bullet \vdash \begin{array}{l}\texttt{case } a \texttt{ of} \\ \quad \texttt{Zero} \to \texttt{Zero} \\ \quad \texttt{Succ } n \to n\end{array} : \texttt{Nat} : \bullet}\ \texttt{case}$$

$$\dfrac{\Delta, a : \texttt{Nat} : \bullet \vdash \begin{array}{l}\texttt{case } a \texttt{ of} \\ \quad \texttt{Zero} \to \texttt{Zero} \\ \quad \texttt{Succ } n \to n\end{array} : \texttt{Nat} : \bullet}{\Delta \vdash \lambda a. \begin{array}{l}\texttt{case } a \texttt{ of} \\ \quad \texttt{Zero} \to \texttt{Zero} \\ \quad \texttt{Succ } n \to n\end{array} : \texttt{Nat} \to \texttt{Nat} : \bullet}\ \lambda - \text{abst}$$

Here, $\Delta = \begin{array}{l}\texttt{data Nat} \to \mathsf{C} = \texttt{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\quad \texttt{Succ} : \mathsf{C} \to \mathsf{C}\end{array}$

$\pi_1 =$

$$\dfrac{\dfrac{\vdots}{\Delta, a : \texttt{Nat} : \bullet, () : 1 : \bullet \ \ \mathsf{context}}}{\Delta, a : \texttt{Nat} : \bullet, () : 1 : \bullet \vdash \texttt{Zero} : \texttt{Nat} : \bullet}\ \text{TypeCons}$$

$\pi_2 =$

$$\dfrac{\dfrac{\vdots}{\Delta, a : \texttt{Nat} : \bullet, n : \texttt{Nat} : \bullet \ \ \mathsf{ctxt}}}{\Delta, a : \texttt{Nat} : \bullet, n : \texttt{Nat} : \bullet \vdash n : \texttt{Nat} : \bullet}\ \text{TermApp}$$

Example C.0.7.

The extended proof of example 7.2.9 is given below:

$$\dfrac{\dfrac{}{\pi_1}\quad\dfrac{}{\pi_2}}{\begin{array}{l}\Delta, \Delta' \vdash \texttt{get}\ i\ \texttt{on}\ \alpha; \texttt{call}\ (\mathit{Counter}(\mathit{add}\ n\ i)(\alpha \Rightarrow) \\ \blacktriangleleft\ (\alpha \triangleright \mathbb{N} \gg X \Rightarrow) \blacktriangleleft \circ\end{array}\quad\begin{array}{l}\Delta, \Delta' \vdash \texttt{put}\ n\ \texttt{on}\ \alpha; \texttt{end}\ \alpha \\ \blacktriangleleft\ (\alpha \triangleright \mathbb{N} \ll \bot \Rightarrow) \blacktriangleleft \circ\end{array}}\ \text{drive}$$

$$\dfrac{\dfrac{}{\Delta}}{\begin{array}{l}\mathsf{drive} \\ \quad \lambda n.\mathit{Counter} :: [\alpha \Rightarrow\ \ ]\ \mathbb{N} \to (\alpha \triangleright \texttt{Ping} \Rightarrow\ \ ) :: \circ := \\ \quad\quad [\alpha \Rightarrow\ \ ]\ \texttt{by}\ \alpha \\ \quad\quad\quad \#\texttt{Ping} : \texttt{get}\ i\ \texttt{on}\ \alpha;\ \texttt{call}\ (\mathit{Counter}(\mathit{add}\ n\ i)(\alpha \Rightarrow)) \\ \quad\quad\quad \#\texttt{Pong} : \texttt{put}\ n\ \texttt{on}\ \alpha; \texttt{end}\ \alpha\end{array}}\ \text{context}$$

$$\Delta = \begin{array}{l}\mathsf{protocol}\ \texttt{Ping} \Rightarrow C_1 = \\ \quad \#\texttt{Ping} :: \mathbb{N} \gg C_1 \Rightarrow C_1\ , \\ \quad \#\texttt{Pong} :: \mathbb{N} \ll \bot \Rightarrow C_1\end{array} \quad \begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \texttt{Zero} : \mathsf{C} \\ \qquad\qquad\quad \texttt{Succ} : \mathsf{C} \to \mathsf{C}\end{array}$$

$$\Delta' = X : \circ, \mathit{Counter} :: \mathbb{N} \to (X \Rightarrow\ \ ) :: \circ, n : \mathbb{N} : \bullet$$

$\pi_1 =$

$$\dfrac{\dfrac{\pi_3}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \\ \vdash (add\ n\ i) : \mathbb{N} : \bullet} \qquad \dfrac{\dfrac{\pi_4}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \ \mathsf{context}}}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) :: \circ, n : \mathbb{N} : \bullet, \\ i : \mathbb{N} : \bullet \vdash Counter :: \mathbb{N} \to (X \Rightarrow \quad) :: \circ}\ \text{ProcessProj}}{\dfrac{\dfrac{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \\ \vdash Counter(add\ n\ i) :: (X \Rightarrow \quad) :: \circ}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \vdash \\ \texttt{call}\ (Counter(add\ n\ i)(\alpha \Rightarrow \quad) \blacktriangleleft (\alpha \triangleright X \Rightarrow) \blacktriangleleft \circ}\ \text{Call}}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) :: \circ, n : \mathbb{N} : \bullet \vdash \\ \texttt{get}\ i\ \texttt{on}\ \alpha; \texttt{call}\ Counter(add\ n\ i)(\alpha \Rightarrow) \blacktriangleleft (\alpha \triangleright \mathbb{N} \gg X \Rightarrow) \blacktriangleleft \circ}\ \texttt{get}\text{L}}\ \text{TermApp}$$

$\pi_3 =$

$$\dfrac{\dfrac{\pi_7}{\begin{array}{l}\Delta, X : \circ, Counter \\ :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \\ \vdash add : \mathbb{N} \to \mathbb{N} \to \mathbb{N} : \bullet\end{array}} \quad \dfrac{\dfrac{\pi_4}{\begin{array}{l}\Delta, X : \circ, Counter \\ :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \ \mathsf{ctxt}\end{array}}\ \text{TermProj}}{\begin{array}{l}\Delta, X : \circ, Counter \\ :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \\ \vdash n : \mathbb{N} : \bullet\end{array}}\ \text{TermApp} \quad \dfrac{\dfrac{\pi_4}{\begin{array}{l}\Delta, X : \circ, Counter \\ :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \ \mathsf{ctxt}\end{array}}\ \text{TermProj}}{\begin{array}{l}\Delta, X : \circ, Counter \\ :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \\ \vdash i : \mathbb{N} : \bullet\end{array}}\ \text{TermApp}}{\begin{array}{l}\Delta, X : \circ, Counter \\ :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \\ \vdash add\ n : \mathbb{N} \to \mathbb{N} : \bullet\end{array}}\ \text{TermApp}}{\begin{array}{l}\Delta, X : \circ, Counter \\ :: \mathbb{N} \to (X \Rightarrow) \\ :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \\ \vdash add\ n\ i : \mathbb{N} : \bullet\end{array}}$$

$\pi_7 =$

$$
\dfrac{
\dfrac{
\dfrac{\pi_4}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet,\\ i:\mathbb{N}:\bullet\ \mathsf{ctxt}\end{array}}
}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet,\\ i:\mathbb{N}:\bullet\vdash\mathbb{N}:\bullet\end{array}}\ \text{TCon}
\quad
\dfrac{
\dfrac{
\dfrac{\pi_4}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet,\\ i:\mathbb{N}:\bullet\ \mathsf{ctxt}\end{array}}
}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet,\\ i:\mathbb{N}:\bullet\vdash\mathbb{N}:\bullet\end{array}}\ \text{TCon}
\quad
\dfrac{
\dfrac{\pi_4}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet,\\ i:\mathbb{N}:\bullet\ \mathsf{ctxt}\end{array}}
}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet,\\ i:\mathbb{N}:\bullet\vdash\mathbb{N}:\bullet\end{array}}\ \text{TCon}
}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet, i:\mathbb{N}:\bullet\\ \vdash\mathbb{N}\to\mathbb{N}:\bullet\end{array}}\ \text{TypImp}
}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet, i:\mathbb{N}:\bullet\\ \vdash\mathbb{N}\to\mathbb{N}\to\mathbb{N}\\ :\bullet\end{array}}\ \text{TypImp}
\quad \textit{new add}
}{\begin{array}{l}\Delta, X:\circ, Counter\\ :: \mathbb{N}\to(X\Rightarrow)\\ :: \circ, n:\mathbb{N}:\bullet, i:\mathbb{N}:\bullet\\ \vdash \mathit{add}:\mathbb{N}\to\mathbb{N}\to\mathbb{N}\\ :\bullet\end{array}}\ \text{TVIntro}
$$

$\pi_4 =$

$$\dfrac{\dfrac{\dfrac{\pi_5}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow) :: \circ, n : \mathbb{N} : \bullet \;\; \mathsf{context}}}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow) :: \circ, n : \mathbb{N} : \bullet \vdash \mathbb{N} : \bullet}\ \text{TypeCons} \qquad new\ i}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow) :: \circ, n : \mathbb{N} : \bullet, i : \mathbb{N} : \bullet \;\; \mathsf{context}}\ \text{TermVarIntro}$$

$\pi_5 =$

$$\dfrac{\dfrac{\dfrac{}{\Delta, X : \circ \;\; \mathsf{ctxt}}\ \pi_6}{\Delta, X : \circ}\ \text{TypeCon} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Delta, X : \circ \;\; \mathsf{ctxt}}\ \pi_6}{\Delta, X : \circ}\ \text{TypeProj}}{\vdash X : \circ}}{\Delta, X : \circ}\ \text{ProcessTyp}}{\vdash \mathbb{N} : \bullet \qquad \vdash (X \Rightarrow \;\;) :: \circ}\ \text{PTypTypAbs}$$

$$\dfrac{\Delta, X : \circ \qquad new\ Counter}{\vdash \mathbb{N} \to (X \Rightarrow \;\;) :: \circ}\ \text{ProcessName}$$

$$\dfrac{\Delta, X : \circ, Counter \qquad :: \mathbb{N} \to (X \Rightarrow \;\;) :: \circ \;\; \mathsf{ctxt}}{\Delta, X : \circ, Counter \qquad :: \mathbb{N} \to (X \Rightarrow \;\;) :: \circ \vdash \mathbb{N} : \bullet}\ \text{TypeCon} \qquad new\ n$$

$$\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \;\;) \;\; :: \circ, n : \mathbb{N} : \bullet \;\; \mathsf{ctxt}\ \text{TermVarIntr}$$

$\pi_6 =$

$$\dfrac{\dfrac{\pi_8}{\Delta \quad \mathsf{context}} \quad new\ X}{\Delta, X : \circ \quad \mathsf{context}}\ \text{TypeVarIntro}$$

$\pi_2 =$

$$\dfrac{\dfrac{\dfrac{\pi_5}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) \quad :: \circ, n : \mathbb{N} : \bullet \quad \mathsf{context}}\ \text{TermProj} \quad \dfrac{\dfrac{\pi_5}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) \quad :: \circ, n : \mathbb{N} : \bullet \quad \mathsf{context}}\ \text{endL}}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) :: \circ, n : \mathbb{N} : \bullet \vdash \mathtt{end}\ \alpha \blacktriangleleft (\alpha \triangleright \bot \Rightarrow) \blacktriangleleft \circ}}{\Delta, X : \circ, Counter :: \mathbb{N} \to (X \Rightarrow \quad) :: \circ, n : \mathbb{N} : \bullet \vdash \mathtt{put}\ n\ \mathtt{on}\ \alpha; \mathtt{end}\ \alpha \blacktriangleleft (\alpha \triangleright \mathbb{N} \ll \bot \Rightarrow) \blacktriangleleft \circ}\ \mathtt{putL}$$

$\pi_8 =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_9}{\Delta'' \quad \mathsf{context}}}{\Delta'' \vdash \mathbb{N} : \bullet}\ \text{TypeCons} \quad \dfrac{\dfrac{\pi_9}{\Delta'' \quad \mathsf{context}}}{\Delta'' \vdash C_1 : \circ}\ \text{TypeProj}}{\Delta'' \vdash \mathbb{N} \gg C_1 : \circ}\ \mathtt{getL} \quad \dfrac{\dfrac{\dfrac{\pi_9}{\Delta'' \quad \mathsf{context}}}{\Delta'' \vdash \mathbb{N} : \bullet}\ \text{TypeCons} \quad \dfrac{\dfrac{\pi_9}{\Delta'' \quad \mathsf{context}}}{\Delta'' \vdash C_1 : \circ}\ \text{TypeProj}}{\Delta'' \vdash \mathbb{N} \ll \bot : \circ}\ \mathtt{getL}}{\begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathtt{Zero} : \mathsf{C} \\ \qquad\qquad\quad \mathtt{Succ} : \mathsf{C} \to \mathsf{C}\end{array} \quad , \quad \begin{array}{l}\mathsf{protocol}\ \mathtt{Ping} \Rightarrow C_1 = \\ \qquad \#\mathtt{Ping} :: \mathbb{N} \gg C_1 \Rightarrow C_1 \quad \mathsf{context} \\ \qquad \#\mathtt{Pong} :: \mathbb{N} \ll \bot \Rightarrow C_1\end{array}}\ \text{ProtocolIntro}$$

where, $\Delta'' = \begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathtt{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , C_1 : \circ \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{Succ} : \mathsf{C} \to \mathsf{C}\end{array}$

$\pi_9 =$

$$\dfrac{\dfrac{}{\begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathtt{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{context} \\ \qquad\qquad\qquad\quad \mathtt{Succ} : \mathsf{C} \to \mathsf{C}\end{array}}\ \pi_{10} \qquad \mathit{new}\ C_1}{\begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathtt{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , C_1 : \circ \quad \mathsf{context} \\ \qquad\qquad\qquad\qquad\quad \mathtt{Succ} : \mathsf{C} \to \mathsf{C}\end{array}}\ \text{TypeVarIntro}$$

$\pi_{10} =$

$$\dfrac{\dfrac{}{C : \bullet \vdash C : \bullet}\ \pi_{11} \qquad \dfrac{\dfrac{}{C : \bullet \vdash C : \bullet}\ \pi_{11} \qquad \dfrac{}{C : \bullet \vdash C : \bullet}\ \pi_{11}}{C : \bullet \vdash C \to C : \bullet}\ \text{Imp}}{\begin{array}{l}\mathsf{data}\ \mathbb{N} \to \mathsf{C} = \mathtt{Zero} : \mathsf{C} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{context} \\ \qquad\qquad\qquad\quad \mathtt{Succ} : \mathsf{C} \to \mathsf{C}\end{array}}\ \text{DataIntro}$$

$\pi_{11} =$

$$\dfrac{\dfrac{\dfrac{}{\varepsilon\ \ \mathsf{context}}\ \text{EmpCtxt} \qquad \dfrac{}{\mathsf{new\ C}}}{C : \bullet\ \ \mathsf{context}}\ \text{TypeVIntro}}{C : \bullet \vdash C : \bullet}\ \text{TypeProj}$$