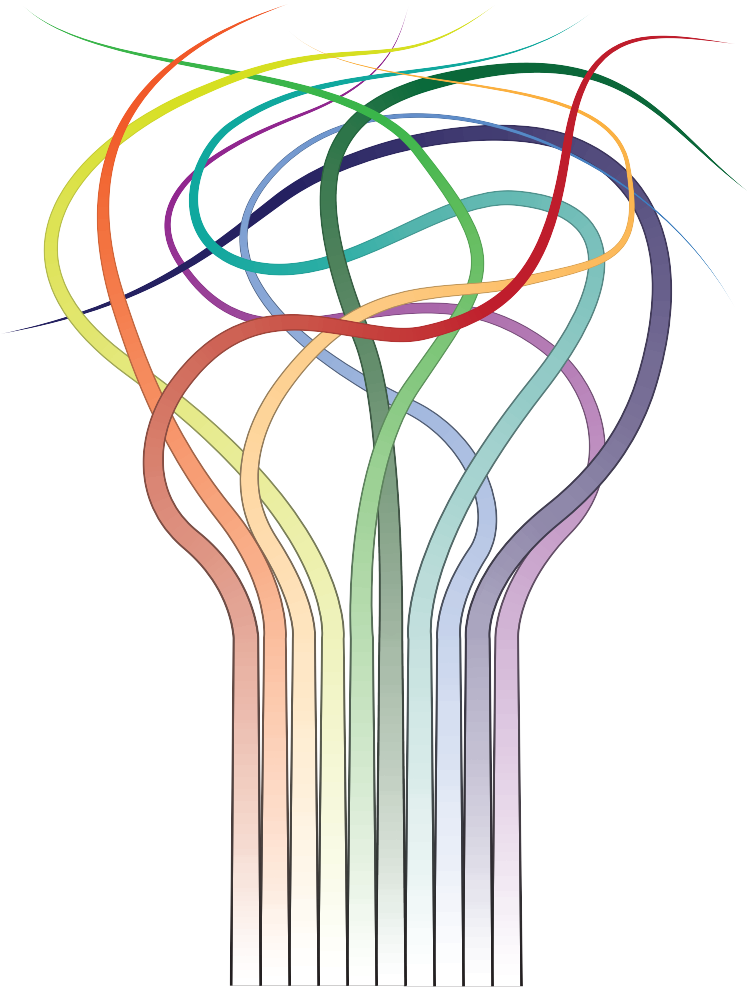
Dependent Types for Invariants in Session Types

Version of December 3, 2018



Wibrand R. van Geest

Dependent Types for Invariants in Session Types

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Wibrand R. van Geest
born in Hoogerheide, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Dependent Types for Invariants in Session Types

Author: Wibrand R. van Geest
Student id: 4189612
Email: W.R.vanGeest@student.tudelft.nl

Abstract

Session types are a formal method to describe communication protocols between two or more actors. Protocols that type check are guaranteed to respect communication safety, linearity, progress, and session fidelity. Basic session types, however, do not in general guarantee anything about the contents of messages, while real-life applications of structured communication, such as money transfers at the ING bank, could benefit greatly from content safety. In this work, we show how to state, verify, and run session-typed protocols with dependent variables in Idris, using Idris' ST library.

We also present an extension to the protocol description language Scribble. Scribble is a language for defining high-level global protocols between multiple actors and comes with a tool that automatically generates type signatures of local protocols for each actor in Java, in a way that these type signatures ensure that local actors follow the global protocol specification. We describe a new variant of the Scribble language which adds support for dependent variables, and a new tool for automatically generating Idris type signatures of local protocols for actors in a way that enforces dependent invariants.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. P. D. Mosses, Swansea University, United Kingdom
Committee Member:	Dr. N. Yorke-Smith, Faculty EEMCS, TU Delft
Committee Member:	J. Bosman, ING
University Supervisor:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft

Preface

From an abstract process problem at ING to a Programming Language thesis, this project has come a long way since its inception. When Joost Bosman assured me that there was a place for a budding Information Architect master student at his ING department, which has since grown into the ING Core Banking University, I never expected that I would end up with Casper B. Poulsen at the Programming Languages group. Casper helped greatly in shaping the ING's need for a statically verifiable communication method into a relevant, state of the art research project. Despite the topic not being my expertise, I feel like the end result is a novel contribution to the field of session types.

I can not overstate the help that Casper provided throughout the project. Without his patience, insight, and guidance the result would be much less impressive, if existing at all. When I was doubtful about the usefulness of the contributions, his passion for the subject reignited my motivation. That is not to say that others did not provide welcome support. Eline has managed to endure me throughout the endeavour, taking care of me when I needed it, and loving me always. My father kept me pointing in the right direction, working and looking forward, as well as providing a healthy kick in the bottom every now and then. My mother, in her infinite patience, ensured that I kept an eye on my own well-being as well. Finally, if all else failed, my friends from Åsene were always ready to grab a beer and forget about the whole thing for a while.

For me, this project has taught me a lot: what I enjoy doing, what I do not enjoy doing (sometimes a little too much for comfort), and where I would like to go. For you, the reader, I hope it will be a learning experience as well, albeit on a more academic level.

Enjoy the read.

Wibrand R. van Geest
Delft, the Netherlands
December 3, 2018

Contents

List of Figures

Chapter 1

Introduction

IT landscapes in big corporations involve interaction and communication between many actors. Ensuring that an implementation of this communication structure is correct is an important aspect of its development, especially in sensitive environments such as banking, where errors in transactions can have severe consequences. Session types (Honda, Yoshida, and Carbone 2008) are a type discipline that statically provide guarantees regarding communication protocols. They ensure that in asynchronous, multi-party protocols: messages are handled in the correct order; no deadlock can occur; all actors adhere to the protocol; and message content is of the expected type. These properties are called *linearity*; *progress*; *session fidelity*; and *communication safety*, respectively[1].

While these properties provide guarantees regarding the *structure* of communication, they provide little information regarding the *content* of messages within that communication. Consider a simple example where a customer requests a bank to transfer an amount of money to a seller. A session typed version of this protocol ensures that the seller will receive *an* amount, but does not ensure whether that amount is equal to the amount initially requested by the buyer. This project introduces invariants to session types to help solve the aforementioned problem.

This improvement to session types is not new. Toninho and Yoshida (2017) provide an extensive theoretical addition of invariants to multi-party session types, but do not (yet) have an implementation of this theory. Wu and Xi (2017) provide the theory and implementation of invariants with session types, but only for binary session types, not multi-party. Instead, this project provides a method for implementing multi-party session types with message content invariance.

The method uses Idris, a dependently typed language, to implement the protocols. Dependent types allow types to become dependent on their values, enabling the static checking of not only content type, but also its value. Idris is, however, a language for which expertise is required. For this project, two actors are considered: (communication) protocol designers, who design and describe the protocol; and programmers, who end up creating the program that represents said protocol. The latter are required for their aforementioned programming expertise, but do not necessarily know the full workings of the protocol. This knowledge needs to be communicated by the protocol designers, and this transfer of information introduces the risk of errors regarding the safety or fidelity of the original protocol. Therefore, a method is desired to not only enable designers to define a protocol, but also make a resulting machine executable implementation.

The goal of this project is therefore to:

> *Support development of declarative machine executable implementations for multi-party protocols with dependently typed invariants.*

[1] See section 6.1 for more information on session types.

This goal breaks down into a number of objectives:

- Safe communication protocols
- Declarative machine-executable protocols
- Enable design by non-programmers

It is important that the protocols are safe. Session types provide a solid basis for this safety, but the goal is to extend such guarantees into the implementation, including the newly introduced invariants. This implementation, or machine-executable protocol, is desired to provide not only a method of protocol design, but also a working program that can be used to execute the results safely. Finally, as much of this process as possible should be automated to ensure that the translation from protocol design to implementation is correct with respect to safety and protocol fidelity. This has led to three contributions:

- A method for implementing session types with invariants in Idris
- ScribbleI, an extension of Scribble including invariants
- A translation tool from ScribbleI to Idris

In chapter 2, a method for implementing multi-party session types in Idris is shown, using an example protocol that will be used throughout the report. This is followed by showing how invariants can be modelled by dependent types in these implementations. Idris was chosen due to its support for dependent types, which can be used to model and statically check invariants, and its ST Library, which provided existing tools to handle stateful programs.

Chapter 3 then introduces Scribble, a protocol design language designed by Honda[2]. This language is much more comprehensible for non-experts and allows for the quick design and checking of new protocols. Additionally, we propose an extension to Scribble, ScribbleI. ScribbleI extends Scribble with support for defining invariants in a protocol. Its syntax is created using SDF3 Vollebregt, Kats, and Visser 2012, a language that provides support for declarative definition of the syntax of programming languages and domain-specific languages.

While the original Scribble program can generate Java code that can be used to create a machine executable implementation, its compatibility with invariants is unclear. Therefore, we developed a method to generate Idris code from a ScribbleI protocol. Currently, this method generates the function signatures and protocol execution in Idris, which ensure that no deviation from either the protocol or invariant specifications is possible. However, while generating parts of the implementation of the functions is feasible, it is not supported by the current version. The code generation is performed by Stratego, an abstract syntax tree transformation language. This process is described in chapter 4.

Chapter 5 concludes the description of the current project by providing an evaluation. This consists of a description of the method of testing throughout the development process, and a more general evaluation of the state of the current project with respect to the desired goals.

In chapter 6, a more detailed explanation of session types is given for those unfamiliar with them. It then provides an overview of related research publications.

The missing pieces are elaborated upon in chapter 7, where a number of desired features are given.

Finally, chapter 8 gives a conclusion of the project.

[2] As described by Yoshida, Hu, et al. 2013

Chapter 2

Implementing Session Types

Session types have been implemented in a number of languages, like Haskell, C, and Java. This project shows that multi-party, asynchronous session types can also be implemented in the dependently typed language Idris[1] (Brady 2013b). Dependent types are types that are, as their name suggests, dependent on values. This property is useful when looking for static methods of checking invariant values. Moreover, Idris provides support for implementing stateful programs through the ST Library, as this chapter describes.

To help with understanding the implementation of a communication protocol, an example protocol is provided in section 2.1. This example is used throughout the report when new concepts are introduced. Section 2.2 then shows how this example can be implemented in Idris. After the reader is familiarized with session types in Idris, the use of invariants in such implementations is explained in section 2.3.

2.1 Running Example

This report will explain a number of different concepts. To help understand these, a running example setting will be used throughout the whole report. The example covers a transaction between a *Customer* who wants to buy an item from a *Seller*. Upon agreement of the transaction, the Customer requests its *Bank* to transfer the required funds to the Seller. Figure 2.1 shows this protocol as a transaction diagram.

First, a *price* request is sent from Customer to Seller (`reqPrice`). This request contains a `String` representing the name of the item the Customer is interested in. The Seller replies with an `Integer` in the message `priceInfo`, providing the price of the requested product. A money transfer is then requested by the Customer from its Bank, with an `Int`eger amount as

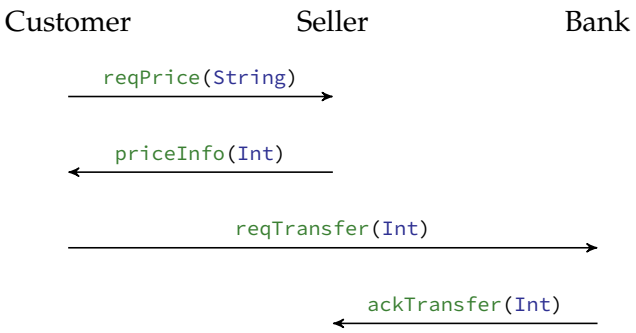


Figure 2.1: A multi-party protocol

[1] Idris (n.d.). URL: https://www.idris-lang.org/.

a parameter, and the Bank acknowledges this transaction by sending the funds to the Seller in `ackTransfer`.

This is the basic example. Later in this report this example will be modified slightly to explain more elaborate concepts, such as presenting an actor with a choice, or introducing recursion. The setting, however, will remain the same.

This protocol can be defined as a session typed protocol. More specifically, it is a *multi-party* session type protocol (Honda, Yoshida, and Carbone 2008), since it covers more than two actors. As Honda, Yoshida, and Carbone (2008) describe, valid session type protocol are guaranteed to have the following properties: *linearity*, *progress*, *session fidelity*, and *communication type safety*.

There is, however, one requirement that the current example does not account for. It should be obvious that the price of the product that is discussed does not change during this interaction, and hence, that the value of the price passed between the actors in the last three interactions should always be the same. However, all that is guaranteed is that the content is of the same type. This thesis presents a solution to this problem.

2.2 Session Types in Idris

This section elaborates on how session types can be implemented in Idris. Idris is a dependently typed languages designed by Brady (2013a), designed from the start to emphasise general purpose programming. Its development is open-source[2], with Brady being the lead contributor and designer.

Implementing a stateful protocol such as session types in a dependently typed language is a challenge in itself. Thankfully, Idris has the ST Library (Brady 2016), an embedded domain-specific language for the specification and implementation of stateful programs. While Brady mentions the library was inspired by session types, the examples in his paper and the tutorial[3] do not provide exact instruction on how to implement them.

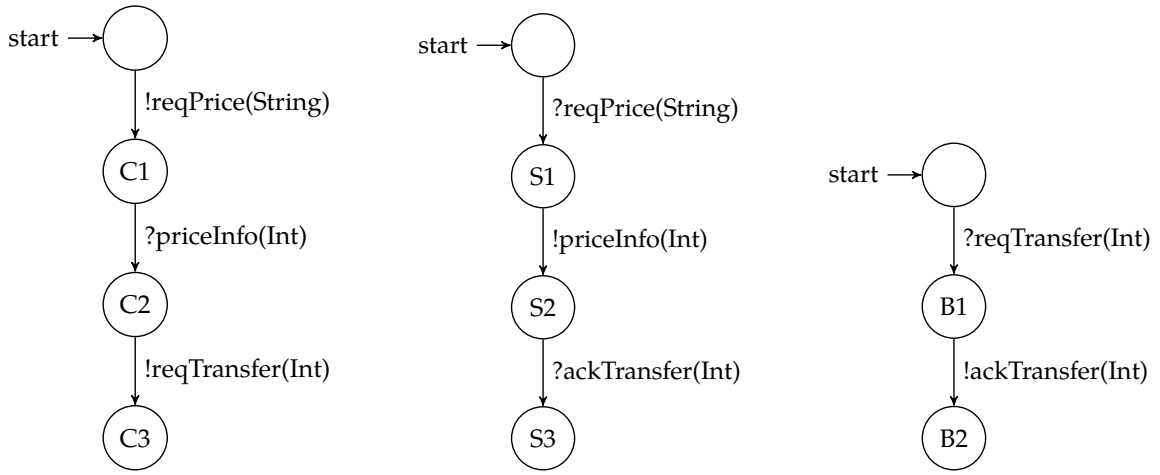
This section is split into four subsections. The first gives a short introduction to *local protocols*, a subset of protocols based on the previously introduced multi-party protocol. Afterwards, the use of the ST Library in this project is explained. An ST based program is broadly divided into three parts: the *interface*, where function signatures are defined; the *protocol*, where these signatures are used to describe the required protocol; and finally the *implementation*, where the functions are given an executable implementation. The subsections in this chapter follow this pattern.

2.2.1 Local protocols

Implementing a multi-party session type protocol first requires generating *local* protocols for each actor, based on the *global* protocol. In short, a local protocol is a version of the protocol from the viewpoint of one actor. How these protocols are generated exactly is discussed in chapter 3. For now, it suffices to know that an actor is only aware of the interactions it is involved in. The state transition system that represents the local protocols of each actor for the running example can be seen in figure 2.2. A sending interaction is denoted with an exclamation mark (!), and a receiving interaction with a question mark (?). The Seller, for example, transitions from the initial state to state S1 upon receiving the reqPrice interaction (?reqPrice(String)). The Idris implementations are implementations of these local protocols. This makes it possible to create three distributed programs, as real life communication protocols would require. In the following sections, the protocol for the Customer is implemented.

[2]Idris on Github (n.d.). URL: `https://github.com/idris-lang/Idris-dev`.

[3]Idris ST tutorial: `http://docs.idris-lang.org/en/latest/st/state.html`



(a) Customer state transitions (b) Seller state transitions (c) Bank state transitions

Figure 2.2: Three transition system specifications for the three actors in Figure 2.1

```
STrans :
  (m : Type -> Type) ->
  (resultType : Type) ->
  (in_res : Resources) ->
  (out_res : resultType -> Resources)
    -> Type
```

(a)

```
ST :
 (m : Type -> Type) ->
 (resultType : Type) ->
 List (Action resultType) -> Type
```

(b)

Figure 2.3: Types of STrans and ST

The protocols for the Bank and Seller can be implemented in a similar manner. Their implementations can be included in the same program (file) by making use of separate *namespaces* to avoid name conflicts, or created in a completely different program.

2.2.2 Interface

Before a protocol can be described in Idris, the function signatures of the interactions need to be specified. This is done in the `interface`. An Idris interface is similar to a Haskell type class or trait in Rust.

The type of stateful computations in the ST library, `STrans`, is given in Figure 2.3(a). Figure 2.3(b) shows the sugared version `ST`. The type `STrans` represents a sequence of actions which can manipulate state. In it, `m` represents an underlying computation context that stateful operations are parameterized by (usually a monad such as the `IO` monad); `resultType` is the result type of the computation; `in_res` is a list of resources available before executing the actions, representing the *pre-state*; and `out_res` is the list of resources available after executing the actions, representing the *post-state*. To avoid the necessity of listing *all* available resources, the `ST` type level function computes this list given a list of *actions*. An `Action` can take several forms, which will be shown in the examples in this section.

Figure 2.4 shows the interface for the `Customer` transition system from Figure 2.2(a). The state transition functions for each interaction is given, along with two functions that start and end the protocol safely: `start` and `done` respectively. The state is given by the `State` dependent type `StateT`. The required resources and resulting state transitions can be seen in the signatures. `reqPrice`, for example, takes `l` and `s` as inputs. `l` is a label variable pointing to the resource that holds the state, referred to as the *stateholder*. `s` is a `String`, in this case

```
1  data State = Ready | C1 | C2 | C3
2
3  interface Customer (m : Type -> Type) where
4    StateT : State -> Type
5
6    start : ST m (Maybe Var) [addIfJust (StateT Ready)]
7
8    done : (l : Var) -> ST m () [remove l (StateT C3)]
9
10   reqPrice : (l : Var) -> (s : String) -> ST m () [l ::: StateT Ready :-> StateT C1]
11
12   priceInfo : (l : Var) -> ST m (Maybe Int) [l ::: StateT C1 :->
13    \p => StateT (case p of
14     Just _  => C2
15     Nothing => C3)]
16
17   reqTransfer : (l : Var) -> (p : Int) -> ST m () [l ::: StateT C2 :-> StateT C3]
```

Figure 2.4: Interface for `Customer`

the name of the product the `Customer` is interested in. The `ST` signature specifies that there is no return type (as indicated by the unit type `()`), and that the resource at label `l` must be in state `Ready` at the moment that this function is called. After the actions are executed, `l` will be in state `C1`.

Transitioning from one state to another is straight-forward, but sometimes this is not sufficient to represent all possible outcomes of a function. One example is handling unexpected communication errors. Even though session types help prevent errors through design, the possibility of network errors still exist. The ST library provides an `Action` form handle this:

```
1  l ::: ty_in :-> (\res -> ty_out)
```

In this form, the resulting state `ty_out` may depend on the function `res`. Figure 2.4 shows this form by augmenting the function signature of `priceInfo` with simple error handling: in case something goes wrong, the state transitions to the final state `C3` and the protocol aborts. This transition is dependent on the result of the return type: the `Maybe` type expects either a `Just Int` value, or `Nothing`. In case of the former, the state transitions to the next state. In case of the latter, it skips to the final state.

Figure 2.4 has shown an `interface` for the `Customer` protocol. The ST library has a larger set of possibilities than shown here, but the provided explanation is sufficient for the implementation of session type protocols. While the interface specifies the state transitions, it does not explicitly specify the order of interactions or handling of variables, nor does it state how to handle the expected variables or what the state changes entail. The following sections explain how this functionality is added.

2.2.3 Protocol

The protocol section of an Idris session type implementation is where the execution order of interactions is specified. It uses the type signatures from the `interface`, as well as additional functions, to set up the protocol and communication between actors.

Figure 2.5 shows the protocol for the running example's `Customer`. The first line lists the interfaces used in this block, in this case `ConsoleIO`, an input/output interface, and the previously described `Customer` interface. The protocol function makes use of the `do`-notation, familiar from Haskell, which is used to sequence operations. It starts with setting up the stateholder `c` using `start` on line 4. Looking back at the function definition of `start` as

```
1  using (ConsoleIO io, Customer io)
2    protocol : ST io () []
3    protocol =
4      do Just c <- start
5           | Nothing => pure ()
6         let product = "Book"
7         reqPrice c product
8         Just p <- priceInfo c
9          | Nothing => done c
10        reqTransfer c p
11        done c
```

Figure 2.5: Protocol for `Customer`

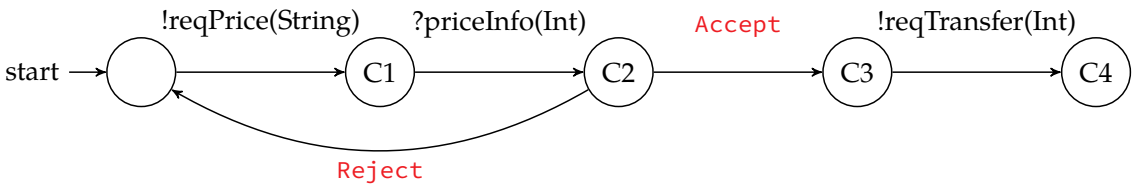
specified in the interface (Figure 2.4), a `Maybe` return type is expected. A `Maybe` returns either a `Just v`, where `v` is a value of the type specified in the definition, or it may return a `Nothing`. This is usually used to represent a function that may fail. Lines 4 and 5 show how each of these is handled. When a `Just` value is returned, this value is assigned to `c` and used throughout the protocol. However, when `Nothing` is returned, `pure ()` is called, a function which returns its argument and is used here to end the protocol. The next line, 6, defines the name of a product. The first interaction of the protocol, `reqPrice`, is a simple call to the function. The next interaction, as we just saw, has the possibility to fail, returning either a `Just` value, or `Nothing`. Lines 8 and 9 show how this is handled. In case of the exception, receiving `Nothing`, the state transitions to the final state and `done` is called to wrap up the protocol. If all goes well, a value is received and assigned to `p`, afterwards continuing the protocol with `reqTransfer`. Since that transitions to the last state, the protocol is done and can also be wrapped up using `done`.

2.2.4 Protocols with choice and recursion

The previous protocols follow a linear flow: one state generally transitions in the one directly following it. In reality, most protocols will not follow a single pattern, but require the possibility of choice. In case of the running example, the Customer should be able to choose whether to accept the given price. Figure 2.6(a) shows the Customer transition system where they can pick a new product after making this choice: after receiving price `p`, the customer can choose to either `Accept` or `Reject` the proposal. When accepted, the protocol proceeds as before. In case the price is rejected, the state reverts to the initial state and the protocol restarts.

Figure 2.6(b) shows the signature for a `choice` function that encodes this pattern. First, the `Accept` and `Reject` options are encoded in a data type, as seen on line 1. As with error-handling, the result state is dependent on a variable: in this case the value of `Choice c`. Should the result be `Accept`, the state transitions to state `C3`. On a `Reject`, the state returns to the `Ready` state, so that the protocol may restart. Figure 2.6(c) shows how this definition is used in the protocol definition on lines 14 to 17. The result of `choice` is stored in `cr`, which is used in a `case` statement to specify what happens for each possible value of `cr`. As explained before, on an `Accept`, the protocol continues by calling `reqTransfer`. With a `Reject`, the protocol is restarted. This is not done by calling back to `protocol`, but instead a new function, `recursion`, is used.

The introduction of this function is required to be able to repeat either a complete protocol or part of it. In both cases it is important that the stateholder, here stored in `c`, remains the same in case of a repeat and is not redefined by calling `start` again. In the case of a partial repeat, only that part which is contained in a recursion block must be repeated. Therefore,



(a) Transition system with choice and recursion

```
1  data Choice = Accept | Reject
2  {- ... -}
3  interface Customer (m : Type -> Type)
       where
4    {- ... -}
5
6    choice : (l : Var) ->
7      ST m () [l ::: StateT (C2) :-> StateT
       (case c of
8        Accept => (C3)
9        Reject => (Ready))]
10
11   {- ... -}
```

(b) Choice type definition

```
1  {- ... -}
2    protocol =
3      do Just c <- start
4           | Nothing => pure ()
5         recursion c
6         done c
7
8    recursion : (l : Var) -> ST io () [l
       ::: StateT {m=io}  Ready :-> StateT
       {m=io} C4]
9    recursion c =
10     do let product = "Book"
11        reqPrice c product
12        Just p <- priceInfo c
13         | Nothing => pure ()
14        cr <- choice c p
15        case cr of
16          Accept => reqTransfer c p
17          Reject => recursion c
```

(c) Protocol handling of `choice`

Figure 2.6: `Customer` has a choice

the recursive block is extracted and defined in a new `recursion` function. Just like any other function, a definition including a state transition is required. A recursive function's state transition adheres to two rules: 1) its pre-state is the same as the pre-state of the first function within its `do`-notation and 2) its post-state is the same as the post-state of the last function within its block. As such, its pre-state in this case is `Ready`, like `reqPrice`. Its post-state is that of `reqTransfer`: `C4`. When a choice is made on line 14, either `reqTransfer` is executed to transfer the state to the final state `C4`, or the `recursion` method is called again to try a different product[4]. This extraction of a recursion block into its separate function is how recursion can be modeled in an Idris protocol.

2.2.5 Implementation

The ST `interface` provides the building blocks for a protocol, ensuring a correct ordering through the state transitions. The `protocol` uses these blocks to define a program. Both of these are hollow, though, until the functions specified in the interface are given meaning in the `implementation`. The Idris ST tutorial[5] provides guidance for implementing stateful functions. Two concepts are extensively used in this project, and thus introduced in this section: *composite resources* and the *socket* library. After these concepts are broadly explained, an example implementation is shown. Before all that, however, *holes* are introduced.

[4] Obviously the product is not changed if it is hardcoded.

[5] `http://docs.idris-lang.org/en/latest/tutorial/index.html`

Holes allow the programmer to partially define a function implementation, then define a hole and have the Idris type checker tell what type that hole is. Not only does this give information as to what behavior is required from the function, but a program with holes can be valid, allowing for the incremental design of a program. In an ST program, holes can be used to check which stateful resources exist and must be altered, deleted, etc. Holes can be defined using a question mark (`?`), followed by a name for the hole. After type-checking, Idris will list the holes and their types. If we would like to know what is required of the implementation of `StateT`, for example, the definition `StateT s = ?StateHole` gives us the following information:

```
1 - + Main.StateHole [P]
2  `-              io : Type -> Type
3                   s : State
4      --------------------------------------
5       Main.StateHole : Type
```

Here we see that the argument `s` is of type `State`. Line 5 tells us the result of `StateT` should be a `Type`.

A state as used with the ST library is thus a definition that holds a stateful type. It provides us with basic types, such as a `State String`, but usually more information is useful to store. This is where composite resources come in. A composite resource is a stateful list of other resources, such as the previously mentioned `State String`. It can be `split` to extract the individual resources, and `combine`d into a single resource again. The states in a session type ST program will most likely be composite resources, since networking in this library requires keeping track of at least one stateful resource: a socket.

The `Socket` library provides the functions to work with sockets. Sockets are stateful resources, which can be in one of the following states: `Ready`, `Bound`, `Listening`, `Open`, and `Closed`. When a socket is created, it is in the `Ready` state. It can then either be bound to a port and wait for incoming connections (through states `Bound` and `Listening`), opening when a socket connects, or it can connect to a listening socket to transition directly to the `Open` state. `Listening` and `Open` sockets can be closed to disconnect them. This relatively simple system enables the use of network communication, essential for session types.

Figure 2.7 shows the implementation of the Customer protocol. It starts with giving the states a type. In this case, the Customer will connect to the Seller and the Bank, and therefore has a `Composite` resource comprised of two sockets. Both sockets should be connected before the protocol is started, so state `Ready` has two `Open` sockets, which is continued up until the final state, `Done`, where the sockets will be `Closed`.

At the start of the protocol, the stateholder must be instantiated. This means that a list of two open sockets must be established. One socket, `sockS`, connects to the Seller, who is waiting for this connection. This is established by creating a socket on line 8 and attempting to connect on line 9. Both of these actions can either succeed (`Right`) or fail (`Left`). In the latter case, the `failure` method throws an exception and the program is aborted. If the connection with `S` is established, the start of a connection with `B` is made by creating a socket, binding it to a port, listening for, and finally accepting a connection, as shown on lines 10 to 13. A new stateful resource is created, called `cust`, and the new sockets `sockS` and `sockB'`[6]are combined into `cust`. The `start` function now has the right resources to be in the `Ready` state: a `Composite` resource containing two `Open` sockets: `sockS` and `sockB'`. Line 18 can therefore return `cust` using the `pure` keyword.

Now that the connections have been established, they can be used for the interactions between actors. Line 20 of figure 2.7 shows the definition of the `reqPrice` function. In this interaction, the customer sends the name of a product to the seller. The two arguments, `cust`

[6]The `accept` function creates a new `Open` socket, rather than changing the listening socket into it. The listening socket is closed and discarded, while the open socket is used for the protocol.

```
implementation (Sockets io, ConsoleIO io, ConsoleExcept String io, Monad io) =>
    CSB_Customer io where
  StateT Ready = Composite[Sock {m=io} Open,Sock {m=io} Open]
  {- ... -}
  StateT C5    = Composite[Sock {m=io} Open,Sock {m=io} Open]
  StateT Done  = Composite[Sock {m=io} Closed,Sock {m=io} Closed]

  start =
      do Right sockS  <- socket Stream | Left _ => failure ""
         Right _      <- connect sockS (Hostname "localhost") 9442 | Left _ => failure ""
         Right sockB  <- socket Stream | Left _ => failure ""
         Right _      <- bind sockB Nothing 9443 | Left _ => failure ""
         Right _      <- listen sockB | Left _ => failure ""
         Right sockB' <- accept sockB | Left _ => failure ""
         cust <- new ()
         combine cust [sockS,sockB']
         close sockB;remove sockB
         putStr "Customer started\n"
         pure cust

  reqPrice cust s =
      do [sockS,sockB] <- split cust
         Right _ <- send sockS s | Left _ => failure "could not send"
         combine cust [sockS,sockB]
         pure()

  priceInfo cust =
      do [sockS,sockB] <- split cust
         Right string <- recv sockS | Left _ => failure "could not receive"
         case (parsePositive {a=Integer} string) of
           Just x => do combine cust [sockS,sockB]; pure x
           _      => failure "parse error"

  choice cust p =
      do [sockS,sockB] <- split cust
         case (p > 50) of
           True =>
             do Right _ <- send sockS "1" | Left _ => failure ""
                Right _ <- send sockB "1" | Left _ => failure ""
                combine cust [sockS, sockB]
                pure Option1
           False =>
             do Right _ <- send sockS "2" | Left _ => failure ""
                Right _ <- send sockB "2" | Left _ => failure ""
                combine cust [sockS, sockB]
                pure Option2

  reqTransfer = ?reqTransfer

  done cust =
      do [sockS,sockB] <- split cust
         remove sockS; remove sockB
         delete cust
         pure ()
```

Figure 2.7: Idris implementation example of the running example's Customer role

and `s`, represent the stateholder and the product name respectively. `s` is simply a string, and `cust` is the previously created composite resource. Line 21 shows how this resource is split into its separate parts, so that the socket to the seller can be accessed. `s` is sent to the seller using the `send` function on line 22, with the Seller socket and `s` as arguments. If this was successful, the two sockets are combined into one resource again, to comply with the definition of state `C1`. In `priceInfo`, the buyer receives the price of the requested product from seller, using the `recv` function on line 28. The received `string` is then parsed to an `Integer x` and returned. At this point, the customer can make the choice to accept the price or not. Line 35 shows this choice based on price `p`. In case the price is too high (`True`), the customer sends that option's number (`"1"`) to both the seller and bank, before returning `Option1` to ensure the right state change. If the price is acceptable, a `"2"` is sent and `Option2` is returned.

Line 47 shows how a hole can be used. There is no implementation for the `reqTransfer` function yet, but the hole ensures the program can typecheck, assuming everything else is fine. If so, Idris provides the type of the hole:

```
1  - + Main.reqTransfer [P]
2   `-                            io : Type -> Type
3                          constraint : Sockets io
4                         constraint1 : ConsoleIO io
5                         constraint2 : ConsoleExcept String io
6                         constraint3 : Monad io
7      --------------------------------------------------------------------------
8       Main.reqTransfer : (l : Var) ->
9      Integer ->
10     STrans io
11            ()
12            [MkRes l (Composite [Sock Open, Sock Open])]
13            (\result => [MkRes l (Composite [Sock Closed, Sock Closed])])
```

Lines 3 to 6 list the interfaces this implementation uses, which every implementation should adhere to. The type of `reqTransfer` starts at line 8. It shows that it requires the `(l : Var)` parameter and an `Integer` as arguments, resulting in an `STrans` type, or an ST state transition. Line 12 shows a list of current stateful resources, based on the implementation of the current state as seen in figure 2.7. There is some unspecified `l`, which is a `Composite` with two `Open` sockets. Seeing as `reqTransfer` is the last interaction in this protocol, the post-state requires not open, but `Closed` sockets, as shown on line 13. Adding the parameters is a quick fix for part of the hole, as is the standard splitting of the resource:

```
1     reqTransfer cust p =
2       do [sockS,sockB] <- split cust
3          ?reqTransfer
```

This provides us with the following hole:

```
1  - + Main.reqTransfer [P]
2   `-          io : Type -> Type
3               sockS : Var
4               sockB : Var
5               cust : Var
6               p : Integer
7               {- ... -}
8      --------------------------------------------------------------------------
9    Main.reqTransfer : STrans io
10         ()
11         [MkRes sockS (Sock Open),
12          MkRes sockB (Sock Open),
13          MkRes cust (State ())]
14         (\result => [MkRes cust (Composite [Sock Closed, Sock Closed])])
```

The first lines show that we now have more resources: the `Vars` `sockS`, `sockB`, and `cust`, gained by `split`ting the `cust` parameter , and the `Integer` `p`. These variables are recognized to be stateful, as seen on lines 11 to 13 . Sockets `sockS` and `sockB` are `Open`, and `cust` has the identity `State` type. The result is to end up with `Closed` sockets. This is easily achieved with the Socket interface's `close` function:

```
1     reqTransfer cust p =
2       do [sockS,sockB] <- split cust
3          close sockS; close sockB
4          ?reqTransfer
```

```
1  - + Main.reqTransfer [P]
2   `-                      {- ... -}
3      --------------------------------------------------------------------------
4       Main.reqTransfer : STrans io
5             ()
6             [MkRes sockS (Sock Closed),
7              MkRes sockB (Sock Closed),
8              MkRes cust (State ())]
9             (\result => [MkRes cust (Composite [Sock Closed, Sock Closed])])
```

The hole now indeed shows that `sockS` and `sockB` are closed. All that is left now is to `combine` the two sockets into the `cust` argument again and return the required return type (`()`), and the function is done:

```
1     reqTransfer cust p =
2       do [sockS,sockB] <- split cust
3          close sockS; close sockB
4          combine cust [sockS,sockB]
5          pure ()
```

This example shows the advantages of programming with holes: one can simply follow the steps listed by the type checker to comply with the pre- and post-states and create a function that adheres to the restrictions in its declaration, without having to fully implementing a function at once. It also shows the limitations of type declarations: this function now does nothing but close some sockets. The *intent*, however, was to send a transfer request to the bank, something that the current implementation completely lacks. The problem can be solved with one line, but this limitation is something to be aware of.

```
1  data State = Ready | S1 | S2 Int | S3 Int
2
3  interface Seller (m : Type -> Type) where
4    {- ... -}
5
6    reqPrice : (l : Var) ->
7      ST m String [l ::: StateT Ready :-> StateT S1]
8
9    priceInfo : (l : Var) -> (p : Int) ->
10     ST m () [l ::: StateT S1 :-> StateT (S2 p)]
11
12   ackTransfer : (l : Var) ->
13     ST m (p' : Int ** p' = p)
14       [l ::: StateT (S2 p) :-> StateT (S3 p)]
```

Figure 2.8: Example of an Idris interface with invariants, for the Seller role

```
1      reqTransfer cust p =
2        do [sockS,sockB] <- split cust
3           Right _ <- send sockB (cast {to=String} p) | Left _ => failure "p not sent"
4           close sockS; close sockB
5           combine cust [sockS,sockB]
6           pure ()
```

The final function used in the protocol is the `done` function. It wraps up all stateful resources and ensures that there are none left at the end of the protocol. Lines 49 to 53 in figure 2.7 show the implementation of this message. It `removes` the resources from the `Composite` resource, and finally `deletes` the stateholder.

2.3 Invariants in Idris

The implementation as discussed in the previous section is still ignorant of the contents of any of the variables. There are no checks to ensure which variables should match, something which the example makes clear can be required for protocols. Toninho and Yoshida (2017) introduced a theoretical framework for adding message content invariants as singleton types. This work proposes a similar idea: the content of messages is represented by a dependent type, which effectively acts as a singleton type. This section will discuss how the addition of invariant affects each stage of an Idris program.

To ensure that the content is invariant for an actor at every point in the communication, it is added as a parameter to its states. Figure 2.8 shows an example of such parameters for the Seller[7]. As can be seen on line 1, some states now have an `Int` parameter, which is added to the states following the introduction of the invariant. In this case price `p` is introduced by the Seller in `priceInfo`, a function that transitions from state `S1` to `S2`. Because of this introduction, `S2` is now parameterized by an `Int`. To ensure this value remains the same throughout the protocol, all following states are parameterized with the same type. Since the parameters are dependent types, their type becomes dependent on the value they are initiated with. In this case, price `p` is used as the value for state `S2`. The type of `S2` is now effectively `S2 p`, rather than the more generic `S2 Int`. This means that `S2 40` and `S2 50` are recognized by the typing system as two different types.

While this parameterization ensures that a single state's type is dependent on the assigned value, it does not mean that the remaining states' types are dependent on that same

[7]The previously used choice function is omitted for brevity.

```
1  data State = Ready | S1 Integer | S2 Integer | S3 Integer String | S4 Integer String Bool
2  {- ... -}
3
4  receiveNew: (l : Var) -> ST m (Integer)
5    [l ::: StateT Ready :-> (\res => StateT (S1 res))]
6
7  receiveKnown : (l : Var) -> ST m (i' : Integer ** i' = i)
8    [l ::: StateT (S1 i) :-> StateT (S2 i)]
9
10 receiveAll : (l : Var) -> ST m (String, (i' : Integer ** i' = i),Bool)
11   [l ::: StateT (S2 i) :-> (\res => StateT (case res of
12     (s,_,_) => (S3 i s))]
```

Figure 2.9: Generic forms of possible receiving interaction function signatures: receiving a new invariant, receiving a known invariant, and receiving a combination thereof

value. For this to be guaranteed, the interface needs to adhere to a simple rule: the parameter value of the end-state of a function must be the same as the parameter value of the start-state of said function. Line 14, for example, shows how the state transition of `ackTransfer` transfers from `S2 p` to `S3 p`, using value `p` for both parameters.

This change to the states affects the functions as well. All function signatures need to adapt their pre- and post-state declarations to include the state parameters. For all but the message interactions, this inclusion is the only change, and thus easily done. Message interactions, however, are affected both in the `interface` and the `implementation`.

2.3.1 Invariants in the Interface

Since messages are the only interactions that influence invariants other than adding parameters to states, only they are discussed. There are two types of message interactions on a local level: sending and receiving interactions.

Line 9 in figure 2.8 shows the sending interaction `priceInfo` for the Seller. With this function, the seller informs the Customer of price `p` and with that, introduces `p` as an invariant. As discussed before, this means that its post-state now includes `p`, whose value must come from somewhere. In the case of sending interactions, a parameter is simply added to the function that specifies this value. The state parameter `p`, associated with `S2`, will take the value of the function parameter `p`. This method also works when an invariant is already known. In that case, the invariant will match values with both the pre- and post-state, ensuring that the sent value equals that of the known invariant.

This is different for receiving interactions. These may receive a single value, which might be a new or known invariant, or a combination of these, possibly with unnamed variables as well.

Figure 2.9 shows example function signatures for these cases. Line 4 shows a way to use an unknown invariant as a state argument. On line 5, the function's return value is captured in `res`. That value, in this case a single `Integer` value, can then be used as an argument for `S1`.

When a known invariant is received, it must be ensured that the received and known value are equal. Line 7 shows a function declaration with a *dependent pair*, written as `(_ ** _)` in Idris. Dependent pairs allow the type of the second element of a pair to depend on the value of the first element[8]. In this case, the pair is used to ensure that the value of `i'`, the return value, is equal to the known invariant `i`, by requiring a proof of this fact from the implementation of the function.

[8] `http://docs.idris-lang.org/en/latest/tutorial/typesfuns.html`

```
1   ackTransfer {p} sel =
2     do [sockC,sockB] <- split sel
3        Right msg <- recv sockB | Left _ => failure "Could not receive ack"
4        case (parsePositive {a=Integer} msg) of
5          Just x =>
6            do case decEq x p of
7                 Yes eq => do close sockC; close sockB; combine sel [sockC,sockB];
8                              pure (x ** eq)
9                 _      => failure "Invariant error"
10         _      => failure "Price not received correctly"
```

Figure 2.10: Implementation of `ackTransfer` for Seller

In case a combination of these is received, the naive method using `\res` does not work anymore, since the return type is now a tuple of return values. The solution is to use a case expression to use the individual elements of the tuple. Line 10 shows that a tuple of values of a new invariant `String`, the known invariant `i`, and unnamed type `Bool` is expected. The case analysis on line 12 names the unnamed `String` and uses it as an argument for state `S3`. The values of the dependent pair and unnamed type are not considered in the case analysis, since they do not interact with the state transition. With this method, all combinations of received values can be handled.

2.3.2 Invariants in the Implementation

In general, the implementation interacts very little with state transitions. The implementation of the states make the rules defined by the transitions concrete, and dictate what resources should be available at the start and end of an implementation. Therefore, there is not much change with respect to the implementation of most interactions.

The most notable difference is in providing the dependent pair that is expected as a return type for interactions receiving a known invariant. The `ackTransfer` function declaration for the Seller on line 12 in figure 2.8 states that a value `p'` is expected, along with the proof that `p' = p`. Figure 2.10 shows the implementation for `ackTransfer`. The first thing to note is that it has a parameter not declared in the function signature: `{p}`. A parameter between curly braces is an *implicit* parameter. It tells Idris to look for a value of `p` within the current context. In this case, the closest parameter `p` will be the one specified as state parameter in the function declaration and, hence, have the invariant's value. This value will be used later in this function. Lines 2 and 3 show how a message `msg` is received using the socket to B. This message is parsed on line 4 using `parsePositive`, a function that expects a number value, in this case explicitly an `Integer`. If the function would have just an `Integer` as a return type, this would be the end of the function. However, a proof showing that this return value equals `p` is required as well. `decEq` on line 6 checks whether the received `Integer x` does indeed equal `p`. If so, it returns `Yes` with the proof of this equality, called `eq` on line 7. After performing the actions to comply with the end-state, in this case closing the sockets, the dependent pair of value and proof can be returned, as seen on line 8. The remaining two lines specify that an error is thrown when either the parsing or equality checking fails.

As mentioned, other interactions have less surprising implementations. Sending interactions that introduce an invariant simply require a value for the invariant as an argument, and receiving interactions either receive simple types or tuples.

2.3.3 Global invariance

As mentioned in 2.2.1, the protocols in Idris are all local: they represent a single actor's perspective. The guarantees regarding invariants are therefore only true for a local protocol, not necessarily for the protocol as a whole. There are two cases to distinguish when considering the global validity: all actors are generated from the same global protocol in the same manner or there is one or more external actors, who adheres to the communication contract (i.e. messages arrive in the specified order), but its internals are unknown. In the first case global invariant holding is guaranteed: each actor generated from a global protocol is sure to have invariants remain unchanged, so all actors collectively do not internally change the invariants and thus, globally, no invariants are changed. This works perfectly in theory, but in practice network errors may occur, so a more robust system is desired. The second case assumes that the external actor adheres to the general structure of the protocol, i.e. the ordering of interactions and reactions thereto, since anything else would make it impossible to execute a session. However, the contents of the message are not guaranteed, nor is the internal knowledge of the external actor. Both cases can be solved by actors passing their local environments along with messages to other actors. The current implementation sends the complete environment with each message, but this could be improved upon by using, for example, blockchain technology. Chapter 7, Future Work, expands on this idea.

2.4 Conclusion

This section showed, step by step, how a simple session type protocol can be implemented in Idris. It used an example protocol, introduced in section 2.1, which describes the interactions between a customer, seller, and bank and will be used throughout the report. Section 2.2 introduced Idris' ST library. Its stateful programming allows for the design of functions with which only a single correct protocol can be created. This protocol is easily designed after the functions are declared, and recursion can be added in the protocol design. In the implementation, the `Sockets` library was used to design the implementation of an interface, with full networking capabilities. In section 2.3 invariants were introduced to the Idris protocol. These affect all aspects of an Idris program differently, and together they assure that invariants remain invariant throughout a local protocol.

When an Idris program properly implements a session type with invariants, it is now impossible to create a protocol and implementation for a single actor that does not adhere to the invariant restrictions, since:

- The invariants stay the same during a state transition.
- When a parameter representing an invariant is used, it is guaranteed to coincide with the state's invariant value
- When a known invariant value is received, the program only continues as normal when said value equals the known value

As such, this implementation of protocols not only provides an implementation for protocols, but ensures that invariant knowledge remains invariant for an actor.

Idris, however, does not provide a way to quickly design and implement global multiparty session types. It introduces language-specific challenges and the management of intricate details that are not associated with generic session types. The next chapter will therefore introduce Scribble (Yoshida, Hu, et al. 2013), a developer-friendly language created specifically to design protocols and check their correctness according to multi-party session type theory.

Chapter 3

Scribble: a developer-friendly representation

Scribble is a simple protocol language designed to easily implement session types. According to Yoshida, Hu, et al. (2013), it was designed by Honda as an alternative to the cumbersome manner of implementing them in existing programming languages. Its syntax is small and simple, the regular session type guarantees are checked, and it is possible to generate Java code from a Scribble protocol, which can be used as a structure for implementing the protocol. Section 3.1 expands on the Scribble language. We illustrate how to use it using the previously introduced running example with the Buyer, Seller, and Bank.

The original Scribble is a useful language for implementing session types as presented by Honda, Yoshida, and Carbone (2008), but as such does not possess the ability to define named variables, let alone invariants. It was therefore necessary to extend the existing Scribble language. Section 3.2 shows how the syntax for invariants was introduced to the Scribble syntax, and discusses how this affects other Scribble features.

3.1 Original Scribble

Scribble's main strenghts are its developer-friendly syntax and its ability to check if a program is a valid session type and, as such, provides the guarantees of session types. This section will introduce the reader to the subset of Scribble's syntax relevant to this project, by implementing the running example in Scribble in section 3.1.1. Section 3.1.2 then expands on the previously mentioned difference between *global* and *local* protocols, and the importance of their relation for multi-party protocols. Section 3.1.3 then explains how Scribble checks the safety of a protocol.

3.1.1 Scribble Syntax

As mentioned before, Scribble is designed to be simple and easy to use. Its syntax reflects this ambition: all basic interactions are represented as intuitive commands, with only a small number of rules to adhere to. Figure 3.1(a) shows a Scribble example, representing the running example of the interaction between Customer, Seller, and Bank (`C`,`S`, and `B`, respectively). Figure 3.1(b) shows the local protocol for one of the actors, which will be discussed in section 3.1.2 and can be disregarded for now.

In Figure 3.1(a), the `module` declaration names the presented module, used for referring to it in other files. The `type` lines are a requirement for the proper functioning of Scribble's Java code generation.

A protocol starts by declaring it, using the `global` or `local`, and `protocol` keywords, followed by the name of the protocol and a list of roles. A role is an actor participating

in the interactions and is denoted using the `role` keyword and a name. The main body of the protocol consists of the interaction sequence; a list of interactions that represent the communication structure. All supported interactions are listed in the Scribble tutorial[1]. A subset of those is used in this project and described here.

The example in figure 3.1 contains three interaction types:

- Message interaction
- Choice
- Recursion

Message interactions are highlighted green in the example, such as `reqPrice` on line 8. They consist of a name, a message payload (which can be empty), and a sender and receiver(s) pair. As the name suggests, these interactions represent the messages passed between actors and are the most basic interactions in a protocol.

A **choice** interaction provides one actor with the ability to choose a course of action. The options are presented as interaction sequences, one of which is executed. In the running example, the choice of the Customer to either accept the proposed price or restart the protocol is encoded in a choice interaction. A choice only has two elements: an actor and a list of interaction sequences. The actor parameter indicates which single actor is responsible for making the choice and thus deciding which interactions are executed. The two or more interaction sequences define the actions to be taken when choosing for that option. They are separated by the `or` keyword and have the same structure as the main interaction sequence, being lists containing any of the above interactions. Line 10 shows a choice followed by two sequences.

A **recursion** declaration consists of two parts: the indicator of the start of the recursion, denoted by `rec`, and one or more references to this declaration using `continue`. On Line 7 the recursion block named `r` is specified. The block to be repeated is encased in curly braces (`{}`). In this case, the whole program is contained in the block, since the whole protocol restarts on a repeat. Line 11 shows that recursion `r` is invoked as one of the choice options, using the `continue` keyword with the name of the recursion, `r`.

3.1.2 Global and Local Protocols

In Scribble, the notions of *global* and *local* protocols exist. This section will explain their differences.

The first iteration of session types, the field of research on which Scribble is based, was *binary*, meaning only two actors were involved in the communication. Only in 2008 did Honda, Yoshida, and Carbone introduce *multi-party protocols*, enabling the design of session type protocols involving three or more actors.

One of the key features that makes this possible is the notion of *projection*: transforming a global protocol, which sees the protocol from an all-knowing perspective, to a local protocol, which only represents the viewpoint of a single actor. Figure 3.1(b) shows the local protocol for our Customer, generated from the global protocol in Figure 3.1(a). Apart from a new name and the `local` keyword, not much has changed in the preamble. The message interactions, however, are now missing an actor. Since a local protocol is described from a single actor's viewpoint, it is a given that one of the parties in an interaction is the local actor. Hence, it is only necessary to describe the relation to the other party using `from` and `to`. Additionally, any interaction that takes place without, in this case, the Customer is outside of its scope and is therefore not present in the local protocol. An example of this can be seen on

[1] Scribble Tutorial (n.d.). URL: http://www.scribble.org/docs/scribble-java.html#SCRIBCORE.

```
1  module Shopping;
2
3  type <java> "java.lang.Integer" from
       "rt.jar" as Integer;
4  type <java> "java.lang.String"  from
       "rt.jar" as String;
5
6  global protocol Shopping(role C, role S,
       role B){
7   rec r {
8     reqPrice(String)      from C to S;
9     priceInfo(Integer)    from S to C;
10    choice at C {
11      continue r;
12    } or {
13      reqTransfer(Integer) from C to B;
14      ackTransfer(Integer) from B to S;
15  }}}
```

(a) Global Protocol

```
1  module Shopping_C;
2
3  type <java> "java.lang.Integer" from
       "rt.jar" as Integer;
4  type <java> "java.lang.String"  from
       "rt.jar" as String;
5
6  local protocol Shopping_C(role C, role S,
       role B) {
7    rec r {
8      reqPrice(String) to S;
9      priceInfo(Integer) from S;
10     choice at C {
11       continue r;
12     } or {
13       reqTransfer(Integer) to B;
14
15  }}}
```

(b) Local Protocol for Customer

Figure 3.1: A simple Scribble example

line 14, where the `ackTransfer` interaction between the Bank `B` and Seller `S` is omitted from the local protocol.

The projection generally follows the rules for projection as described by Honda, Yoshida, and Carbone (2008). Informally, they can be described as:

- If the actor takes part in an interaction, that interaction is added to the protocol as either a sending or receiving interaction(`to` or `from`)
- If the actor is involved in a choice option, the choice and the projection of all options is included in the protocol
- If the actor in involved in a recursion block, the recursion and projection of said block is added to the protocol

Using these rules, almost any global protocol[2] can be projected unto a set of local protocols for each actor. Since these protocols define the exact interactions one actor can expect to send and receive, without knowledge outside its scope, these local protocols can be used to create the separate, distributed programs that, together, represent the protocol as a whole.

The syntax ensures that a protocol description makes some basic sense, and the projection rules guarantee that a valid global protocol can be used to generate valid local protocols. However, these rules are not enough to ensure that a protocol is valid in itself. The next section describes some of Scribble's checks to provide this assurance.

3.1.3 Protocol Safety

Session types and their advantages have been named a number of times, and this section will expand on these concepts and their effects on Scribble.

As described in the introduction, session types offer four guarantees regarding communication:

[2]One rule regarding parallel compositions is omitted, since it is not used in this project.

- **Linearity** guarantees that all messages are handled in the specified order, even in an asynchronous setting.
- **Progress** means that all valid protocols are deadlock-free.
- **Session fidelity** says that all actors adhere strictly to the protocol as defined.
- **Communication safety** guarantees that no errors will occur due to unexpected messages or payload types.

These are powerful guarantees, especially in a multi-party asynchronous setting. Even better is the fact that Scribble can check for these properties automatically. In order to benefit from session type properties, a number of rules need to be adhered to. The Scribble documentation[3] provides information regarding the more abstract rules required to adhere to session type wellformedness, but there is one relevant rule that we will encounter in this project that requires some attention.

This rule pertains to *choices*, and is discussed since it will affect a design choice for the translation described in the next chapter. After a choice is made, actors must be *enabled* before being able to act. The actor making the choice is the only one enabled by default, and other actors can be enabled by receiving a message from an enabled actor. This is based on the notion that an actor is unaware of the result of the choice until it receives a message, confirming which branch has been picked. Consequently, the interaction type (sending or receiving) and peer (sender or receiver) of an enabling interaction must be the same *in all choice branches*. This will be explained using the running example which, in fact, does not adhere to these rules.

Figure 3.2(a) shows the global protocol as previously seen. When a choice is made, in this case by the Customer, the other actors do not know which message is to be expected, since they are unaware of the result of the choice until they are enabled. When the first choice option is chosen, the protocol is repeated and Seller `S` can expect to receive the `reqPrice` message from Customer `C`, as seen on line 8. However, when the second option is chosen, the first message it would receive would be the `ackTransfer` message from the Bank `B` from line 14. As such, `S` does not know whether it will receive a message from `C` or from `B`, breaking communication safety.

There are multiple ways to deal with these situations. One is trying to redesign the whole process. A much simpler solution is to use *dummy* messages in one branch such that all choice enable the actors in the same order. Figure 3.2(b) shows this solution. In both branches, `S` now expects an incoming message from `B`. Depending on which message is received, `dummy2` or `ackTransfer`, it knows which branch has been chosen by `C`.

When this and the other rules are adhered to, Scribble can verify whether a protocol is and the session types guarantees hold. None of these guarantees, however, provide information regarding the content of messages, other than their type. The next section will show how Scribble is extended to allow the specification of more detailed payload information and how this affects Scribble's functionality.

3.2 ScribbleI: Invariants in Scribble

Chapter 2 explained how session types can be implemented in Idris and how invariants can be added to them. The previous section showed that Scribble is capable of describing session typed protocols, and now it is time to show how invariants are added to it.

Initially, only a simple syntax extension is required. Section 3.2.1 describes the process of introducing the new syntax to the existing set. It should be clear by now, however, that

[3] Scribble Tutorial (n.d.). URL: `http://www.scribble.org/docs/scribble-java.html#SCRIBCORE`.

```
module Shopping;

type <java> "java.lang.Integer" from
    "rt.jar" as Integer;
type <java> "java.lang.String"  from
    "rt.jar" as String;

global protocol Shopping(role C, role S,
    role B){
 rec r {
   reqPrice(String)     from C to S;
   priceInfo(Integer)   from S to C;
   choice at C {
     continue r;
   } or {
     reqTransfer(Integer) from C to B;
     ackTransfer(Integer) from B to S;
}}}
```

(a) Global Protocol

```
module Shopping;

type <java> "java.lang.Integer" from
    "rt.jar" as Integer;
type <java> "java.lang.String"  from
    "rt.jar" as String;

global protocol Shopping(role C, role S,
    role B){
 rec r {
   reqPrice(String)     from C to S;
   priceInfo(Integer)   from S to C;
   choice at C {
     dummy()  from C to B;
     dummy2() from B to S;
     continue r;
   } or {
     reqTransfer(Integer) from C to B;
     ackTransfer(Integer) from B to S;
}}}
```

(b) Valid Global Protocol

Figure 3.2: Enabling Actors

Scribble is more than a syntax. Such an extension has consequences for its other features. Section 3.2.2 describes how projection was affected. Section 3.2.3 describes the simple solution to retaining Scribble's sophisticated validity checking features.

3.2.1 Syntax

An extension to an existing syntax can take several forms. In this case, Scribble's syntax is implemented in ANTLR 2[4], a deprecated language. Since this project is designed in Spoofax[5] and ANTLR 2 is an outdated format, the choice was made to reimplement a subset of Scribble's syntax in SDF3 (Vollebregt, Kats, and Visser 2012). This subset, later extended with invariant support, is called ScribbleI.

Scribble's syntax can be divided in three sections: lexical syntax, containing keywords, whitespace handling, etc.; structural syntax, covering module names, imports, and type declarations; and lastly the syntax for operations, where the protocol specifications and interactions are defined. The first is straightforward and not covered in this report, just like the structural syntax, which is useful to know for the creation of protocols, but is also rather simple to implement. The syntax of operations, however, is explained in more detail in this section.

Figure 3.3 shows the structure of the syntax as implemented in SDF3. Here, the angle brackets (`< ... >`) mean a *template* is defined for the constructor on the left-hand side. In a template, symbols are either placeholders or literal strings, where placeholders are enclosed in angle brackets within the template. Placeholders are of the form[6]:

- `<Sort?>` : Optional placeholder

[4] ANTLR (n.d.). URL: http://www.antlr.org/.

[5] Spoofax (n.d.). URL: http://www.metaborg.org/.

[6] SDF3 Reference Manual: http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/reference.html

```
1  Module.Mod =
2    <<Moduledecl>
3  <{Protocoldecl "\n"}*>>
4
5  ProtocolDecl.GProtDecl     = <<GProtHeader> <GProtDefinition>>
6  GProtHeader.GSimplePHeader = <global protocol <ProtocolName> <RoleList>>
7
8  RoleList.Roles = <( <{Roledecl ", "}+> )>
9  Roledecl.Role  = <role <Rolename>>
10
11 GProtDefinition.GPDef = GProtocolblock
12
13 GProtocolblock.GPBlock       = <{<GInteractionsequence>}>
14 GInteractionsequence.GIntSeq = <<{GInteraction "\n"}*>>
15
16 GInteraction.GIMsgTransfer = GlobalMessageTransfer
17 GInteraction.GIChoice      = GlobalChoice
18 GInteraction.GIRecursion   = GlobalRecursion
19 GInteraction.GIContinue    = GlobalContinue
20
21 GlobalMessageTransfer.GTrans = <<Message> from <Rolename> to <{Rolename ", "}+>;>
22 GlobalChoice.GChoice         = <choice at <Rolename> <{GProtocolblock "or"}+>>
23 GlobalRecursion.GRecursion   = <rec <RecursionName> <GProtocolblock>>
24 GlobalContinue.GContinue     = <continue <RecursionName>>
25
26 LocalMessageTransfer.LRecv   = <<Message> from <Rolename>>
27 LocalMessageTransfer.LSend   = <<Message> to <Rolename>>
28
29 Message.MessagePlID = <<MessageName>(<Payload?>)>
30 Payload.Load = <<{Payloadelement ", "}+>>
31 Payloadelement.PLType = <<TypeName>>
32
33 Payloadelement.PLDeclaration = <<VarName> : <TypeName>>
34 Payloadelement.PLName = <<Payloadname>>
```

Figure 3.3: ScribbleI syntax in SDF3. The highlighted lines at the bottom are the new syntax additions

- `<Sort*>` : Repetition (`0...n`)
- `<Sort+>` : Repetition (`1...n`)
- `<{Sort "s"}*>` : Repetition with separator `s`

As a note, most of the presented syntax is for global protocols, indicated by a `G`. Unless otherwise stated, its local equivalent also exists, but is omitted for brevity. Anything not annotated with a `G` or `L` is common for both types.

At the root of the syntax is a declaration, which contains a header and the definition of the protocol, represented by a `GSimplePHeader` and `GProtDefinition` respectively. The definition is simply an interaction block, which line 14 shows is a list of interactions. These interactions can take the familiar forms of message interactions, (lines 16 and 21), choices (lines 17 and 22), recursion declarations (lines 18 and 23), and continue statements (lines 19 and 24). Lines 26 and 27 show the local variants of message interactions. Choices and recursions have one or more interaction blocks as a parameter. As seen in the syntax, these are equivalent to the main interaction block and treated as such during, for example, projection. As such, they act as the roots of their own branches. Message interactions do not have such

expansive children, and can be seen as leaf nodes. Messages can carry a `Payload`, as seen on lines 29 to 31. This payload is an optional list of one or more payload elements.

So far, nothing is inherently different from the existing Scribble syntax. Recall that what is needed is the possibility to express that the payload of one message should be equal to the payload of another. To accommodate this, the naming of payloads is added to the syntax. These additions to the original Scribble syntax are shown on lines 33 and 34, highlighted in yellow. In addition to using generic types as a payload, payloads can now introduce and use named payloads.

Lines 1 to 31 thus show a subset of the original Scribble language. Lines 33 and 34 show the only addition to the syntax required to allow for the use of named invariants.

3.2.2 Projection

While what is presented is a syntactically tiny extension, it affects certain Scribble functionality. Especially the projection of global to local protocols in a multi-party environment becomes important. In addition to the projection rules mentioned in section 3.1.2, the projection for the extended syntax requires taking into account the new invariants. For those, the following restrictions are followed:

1. Invariants' names must be unique within a protocol
2. Invariants must be declared before use
3. An actor cannot reference an invariant unless it has *knowledge* of it
4. Invariants are limited to their scope

These restrictions must be true for both global and local protocols. From restriction 1 it follows that in a global protocol, an invariant can be declared only once. The second restriction is a matter of course and almost trivial to check in a global setting. However, as the example afterwards will show, this is an important rule to ensure during and after projection as well, especially in multi-party environments. The third restriction mentions *knowledge*. Actor knowledge is a set of invariants known to an actor. Each actor may only reference an invariant that is known. Knowledge is gained by either declaring an invariant, or receiving the invariant from an actor that has knowledge of it. This restriction ensures that an actor unfamiliar with the value of an invariant can not reference it and, as such, only one value for an invariant is propagated to each actor. The last restriction is designed to make sure that an invariant declared within a limited scope is not used outside of it. This scope can be limited by, for example, being declared in a choice or recursion block. These restrictions are enforced by creating an *environment*, consisting of the knowledge sets of all actors. The details of this environment can be found in section 4.1: Environment.

To illustrate these restrictions and their importance during projection, the projection for the running example's actor Bank `B` is described. Figure 3.4(a) shows the running example's protocol extended with the named invariant `p`, representing the price of the requested item. It is declared on line 6 using the `(Name : Type)` syntax. Now, the interactions requesting and confirming the money transfers to and from the Bank can reference `p` in order to ensure that they are the same value as the original declaration. Note that this declaration involved both the Seller `S` and Customer `C`, but not the Bank. A naive projection retaining the payloads precisely results in the protocol in Figure 3.4(b). There, the Bank receives reference `p` from the Customer. In this protocol, restriction 2 is not true: Bank was never introduced to `p`. One solution could be to broadcast a declaration to all actors. However, since actor knowledge is already kept track of, it is known that `reqTransfer` is the first time that Bank has come in touch with `p`. For the local protocol, `reqTransfer`'s payload is therefore updated to become a declaration of `p`: `(p : Integer)`. The Bank's local protocol now complies with all restrictions

```
module Shopping;

global protocol Shopping(role C, role S,
    role B){
 rec r {
   reqPrice(String)     from C to S;
   priceInfo(p : Integer)   from S to C;
   choice at C {
     dummy()  from C to B;
     dummy2() from B to S;
     continue r;
   } or {
     reqTransfer(p) from C to B;
     ackTransfer(p) from B to S;
}}}
```

(a) Global Protocol with Invariants

```
module Shopping_B;

local protocol Shopping_B(role C, role S,
    role B){
 rec r {


   choice at C {
     dummy()  from C;
     dummy2() to S;
     continue r;
   } or {
     reqTransfer(p) from C;
     ackTransfer(p) to S;
}}}
```

(b) Local B Protocol with Invariants

Figure 3.4: Projection with Invariants

and is a safe protocol. This new declaration is safe to add since, on a global level, restriction 3 ensures that the actor introducing another actor to a new invariant has the same knowledge of the invariant as its originator.

3.2.3 Validity Checks

Scribble's validity checks for communication protocols are one of the main reasons to use it. Since the syntax of the extended Scribble violates that of the original, it is not possible to use Scribble to check the validity of protocols with invariant declarations. Therefore, a tool was created that removes the new syntax from a protocol, so that it is a syntactically valid Scribble protocol. Scribble can then be used to check whether the protocol is a valid session type.

The tool works by creating placeholder type definitions for the types encountered in the protocol and replacing the invariant declarations and references with the invariant's type. Since all other syntax is a subset of Scribble's original syntax, all other constructs can simply be pretty-printed.

This tool was tested by generating Scribble files from the test cases mentioned in 5.1.2, both for the global and local protocols, and type checking those with the Scribble program. Additionally, the projection function was tested by generating a Scribble global protocol from a ScribbleI protocol, projecting both using their respective tools and comparing the results. This comparison was done by generating a Scribble local protocol from the result of the ScribbleI projection and comparing it to the original Scribble's projection, as shown in figure 3.5.

3.3 Conclusion

This chapter introduced Scribble, a simple protocol design language. Its syntax is easy to pick up, yet it can provide strong guarantees for protocols designed with it.

When invariants are added to Scribble, this does not only affect the syntax, but also projection and the inherent session type features. It was explained what the impacts of these effects is and how they are handled.

The reader is now familiar with two ways of designing protocols: one is an implementation in Idris, which provides a machine executable program, but does not provide global

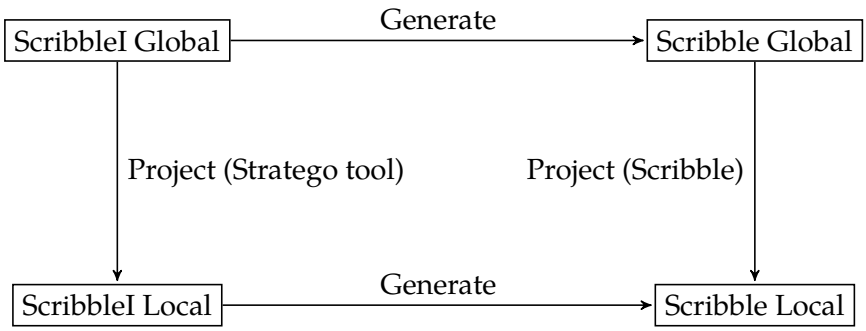


Figure 3.5: Verifying ScribbleI to Scribble transformation tool

guarantees, and it can be somewhat cumbersome. On the other hand, Scribble is a developer-friendly language that allows for quick prototyping, but does not offer an implementation with invariants. It is therefore time to bring those together. The next chapter, chapter 4, shows how Idris code is generated from a Scribble protocol.

Chapter 4

Generating Idris code from Scribble

Scribble and Idris each have their own strengths and weaknesses, several of which are each others mirror opposites. An ideal situation would therefore be if it is possible to profit from both. This chapter describes how the larger, more intricate Idris code can be generated from a smaller, simpler Scribble protocol.

Idris and Scribble syntax and semantics differ greatly. Generating one from the other therefore requires multiple steps. The first step is to generate local protocols from global protocols, since those represent a single actor's protocol, enabling a distributed implementation of each individual actor. Chapter 3 covered how this could be done. The next step is to gather the information regarding states and invariants that is required to correctly parameterize each function. Then the translation can happen. The translation is performed using a small, specific syntax representing the relevant parts of Idris' syntax. Like Scribble's syntax, this was created using SDF3. Having both languages available in the Spoofax workbench allows the use of Stratego to perform the translation. Stratego is a language for program transformation and is used to transform the AST of a Scribble protocol into an AST of the intermediary syntax. This transformation happens separately for each of the three Idris ST program parts: the interface, the protocol, and the implementation. Finally, Idris code is generated by using Spoofax' built-in pretty-printer.

This section describes each of the steps in detail. First, the intermediary syntax is described in section 4.2. It presents the formal definition and resulting Idris code of the new syntax. After this, the reader is shown how the environment, which keeps track of states and invariant scope, is generated in section 4.1. This environment is used extensively in other parts of the translation, making it a crucial aspect, even in its relative simplicity. Then the code generation for the interface and protocol parts of the code generation are discussed in sections 4.3 and 4.4, respectively. The automatic generation of implementation details for the signatures in the interface proved to bring more challenges than expected. Section 4.5 discusses these challenges and how the now missing generation of this part could be designed.

4.1 Environment

All parts of the desired Idris code are highly dependent on their context: the function specifications in the interface must have the correct state transitions assigned, along with the possible invariants associated with those states; the protocol needs to list the methods in the correct order, dealing with choice branches and recursions; and the implementation must use the appropriate parameters. As such, the first step in the code generation is collecting the necessary information in an *environment*.

The two main pieces of information required for the code generation are the states and the invariants associated with each of them. Since this data is related, they can be stored in the same structure. A list of lists is used to represent the state space, with different interactions affecting the state space differently.

The environment is created for a specific actor, since an environment is dependent on the actor associated with it: an interaction in which an actor does not take part should not affect that actor's environment. This could be done by using the actor's local protocol as a basis for generating the code. However, a local protocol might miss important information. Figure 3.4, for example, showed that Bank `B` receives price `p` for the first time with the `reqTransfer(p)` interaction. Since `p` is already declared in the global protocol, its type is not sent with the payload. However, as `B` was not involved in the initial declaration, it is missing in the naively projected protocol in Figure 3.4(b), and an environment generated from this local protocol would not be able to infer the type of `p`.

The environment generation is therefore done at a global level, but does not create one global environment. Instead, the local environments for each actor are simultaneously generated, so that these can be directly used in the projection. The result of the environment generation is therefore a set of local environments, with the global information used to ensure a number of restrictions:

1. Invariants' names must be unique
2. Invariants must be declared before use
3. An actor can not reference an invariant unless it has knowledge of it
4. Invariants are limited to their scope

These rules are enforced using the actor *knowledge*. Actor knowledge is the set of invariants and their types of which an actor is aware. It is represented by a tuple with the name of the actor, and a list of known invariants and their types. Each time an actor is involved in an interaction, its knowledge may be updated, depending on the information introduced by the interaction and whether the actor takes part in it.

Restriction 1 is checked by scanning the combined knowledge of all actors when a new invariant is declared. If any actor has knowledge of an invariant with the same name, the restriction does not hold. Similarly, all knowledge is scanned to find information about an invariant when it is referenced. If no actor has knowledge regarding a referenced invariant, it has not been declared and restriction 2 is not true. A single actor's knowledge is checked to ensure the third restriction. Finally, the last restriction holds by construction, as will be seen in the rest of this section.

The rest of this section is split in two. First, the individual Stratego rules for updating the environment for each interaction will be discussed: messages, choices, recursions, and continues. Afterwards, the result of these rules applied to the running example is discussed.

4.1.1 Building the environment

Each interaction type has its own unique influence on the environment: Message interactions may introduce or receive new invariants, choices create branches in a program, and recursion and continue statements introduce interaction blocks that can be looped. When the environment is updated, either a list of lists, representing a block of states, or a list of tuples, representing a single state, is added to each actor's environment. A list of states represents the states for a block of interactions, like a choice block or recursion block. The updates to the environment caused by interactions within such a block are applied to the new

[1] `Integer` is shortened to `Int` in the examples

```
/* ... */
rec r {
  reqPrice(String) from C to S;
  priceInfo(p : Integer) from S to C;
  choice at C
  {
    dummy() from C to B;
    dummy2() from B to S;
    continue r;
  } or {
    reqTransfer(p) from C to B;
    ackTransfer(p) from B to S;
  }
}
/* ... */
```

(a) Scribble Protocol

```
[
 [
  [],
  [],
  [(p,Int)],[
  [
   [(p,Int)],

   [(p,Int)]
  ],[
   [(p,Int)]

  ]]
 ]
]
```

(b) Customer

```
[
 [
  [],
  [],
  [(p,Int)],[
  [

   [(p,Int)],
   [(p,Int)]
  ],[

   [(p,Int)]
  ]]
 ]
]
```

(c) Seller

```
[
 [


  [],[
  [
   [],
   [],
   []
  ],[
   []
   [(p,Int)]
  ]]
 ]
]
```

(d) Bank

```
[ ... ,[("C",[(p,Int)]),("S",[(p,Int)]),("B",[])],[[("C",[(p,Int)]),("B",[])], ... ], ...]
```

(e) `intKs` for lines 5 to 7

Figure 4.1: Environment Generation for Customer, Seller, and Bank[1]

list. A single state for a single actor consists of a list of tuples, with each tuple representing invariant knowledge: its name and type. As such, when a new state is added to an actor's environment, such a list of tuples is added to an actor's environment. These states represent the pre-states of the interaction that spawned them. Figure 4.1 shows an example of such environments for the actors of the running example. In a later stage, all states are named for use in the Idris code.

The transformation is carried out using Stratego, an abstract syntax tree transformation language. Stratego uses *rules* to apply a transformation. Rules transform the left hand side of a rule into its right hand side. A rule's name is not unique, but may have multiple versions with different left hand side arguments. When one matches the input, it is applied. To allow for more complex transformation, the `where` and `with` keywords may be used. `where` is followed by a number of strategies that should succeed for the rule to succeed. If it fails, the rule fails and an attempt is made to use another rule. `with` is similar, but instead of trying to apply another rule, the program is terminated. Within these strategies, assignments of the form `var := S` can be defined, as well as conditionals. Conditionals take the form `C < S1 + S2`, where `S1` is executed if `C` is true, and `S2` otherwise. With the following rules, a named variable is often checked to contain a certain value or pattern. This is done using the `?` (question mark), followed by the desired pattern and the strategy or variable to be tested. For example, `<?[]> x` tests whether `x` is the empty list `[]`.

This section discusses how the environment is built by visiting each interaction, and what each type of interaction add to the environment being built.

Message Interactions

Message interactions are, from a state change point of view, almost trivial: they always transfer from one state to the next. They are, however, the only interactions that make use of invariants, which directly affects the states from and to which they transition. It also means that the checks for the previously mentioned restrictions are enforced when message interactions are encountered during environment generation.

```
1  create-environments: ([GMsgTransfer(msg,sender,receivers) | ints], actorKs) ->
2      [intKs | <create-environments> (ints, actorKs'')]
3  with
4    intKs := <create-intKs> (actorKs, [sender|receivers])
5  ; invs  := <collect-all(?VariableDecl(_,_))> msg
6  ; refs  := <collect-all(?PLName(_))> msg
7  ; (not(<?[]> invs)
8    < actorKs' := <add-invariants> (invs, [sender|receivers], actorEs)
9    + actorKs' := actorEs)
10 ; (not(<?[]> refs)
11   < actorKs'' := <add-invariants> (refs, [sender|receivers], actorKs')
12   + actorKs'' := actorKs')
```

Figure 4.2: Stratego rule for message interaction environment creation

Since message interactions transition from one state to the next, a single new state is introduced for them. This is done by adding a list of known invariants to the environment being built. Recall that states in the environment represent the *pre-state* of interactions. This means that new invariants introduced by the current interaction, either by sending or receiving them, does not affect the state added to the state space. The involved actors' knowledge, however, should be updated, so that following states do take into account the new invariants. For a message interaction, therefore, a generation rule should introduce a new state to each involved actors' environment with each of their respective 'old' knowledge, and each actors' knowledge should be updated for following interactions.

Figure 4.2 shows the rule that applies this logic and generates the environment for a message interaction. Line 2 shows that the result is a list whose head is the newly added state for each actor, `intKs`, and the tail to the rest of the environment to be generated, using the remaining interactions.

The list of known invariants representing the new state is generated using the `create-intKs` rule. For message interactions, `create-intKs` returns the current knowledge sets of those actors involved in the interaction, ensuring that the environment of actors not involved is not changed. This set of knowledge is added to the environment, representing a state for each of the local environments. An example of such a knowledge set, Figure 4.1(e) shows an excerpt of the raw environment additions, of which Figures 4.1(b) to 4.1(d) are formatted versions created by extracting a specific actor's environment.

The remainder of the rule has to do with the possible updates to the actor knowledge this message interaction may induce. Lines 5 and 6 collect the invariant declarations and references present in the payload of this message. Lines 7 to 12 specify that if invariants or references exist, then add them to the set of actor knowledge using the `add-invariants` rule (discussed hereafter). This (possibly) updated set of actor knowledge is used as an argument for creating the environment for the successors of this interaction.

The `add-invariants` rule, shown in figure 4.3, adds invariants to each actor's knowledge, if required. Additionally, this is also where the restrictions mentioned before are checked. The rule on line 1 covers the case for payload declarations. First, it checks whether the invariant is known for any actor, by looking for its name in the set of all actors' knowledge. If the name already exists, the rule fails with an error message. If the name is new, the actor knowledge set is updated using the `add-to-Ks` rule, which simply adds the supplied invariant to the knowledge set of the provided `actors`.

Line 9 shows the rule for invariant references. This rule starts with finding the knowledge of the actor, which is then used on line 12 to both check if the sender has knowledge of an invariant with name `ref` and find its type. The latter is required because the message only carries a reference to, not the type of the invariant. If no such knowledge can be found for the

```
add-invariants: ([PLDeclaration(var,type)|invs], actors, actorKs) -> newactorKs
with
  (not(<?[]> <collect-all(?(var,_))> actorKs)
      < <debug> ["Duplicate invariant declaration for ", var];fail
      +
; actorKs'   := <add-to-Ks> ((var,type), actors, actorKs)
; newactorKs := <add-invariants> (invs,actors,actorKs')  )

add-invariants: ([PLName(ref)|refs], [sender|receivers], actorKs) -> newactorKs
  with
    senderE := <lookup> (sender, actorKs)
  ; (<?[(ref,refType)|_]> <collect-all(?(ref,_))> senderE
    < actorKs' := <add-to-Ks> ((ref,refType),receivers,actorKs)
    + <debug> ["Unknown reference ", ref, " in message for sender ", sender]; fail)
  ; newactorKs := <add-invariants> (refs, [sender|receivers], actorKs')
add-invariants: ([],_,actorKs) -> actorKs
```

Figure 4.3: `add-invariants` rule. The first rule adds invariant information to an actor's environment if said invariant does not already exist. The second rule does the same for references to known invariants.

```
create-environments: ([GIChoice(_,blocks) | ints], actorKs) ->
    [intKs, blockRes | intRes]
with
  intKs    := <create-intKs> (actorKs, blocks)
; blockRes := <create-environments> (blocks,intKs)
; intRes   := <create-environments> (ints,actorKs)

create-environments: ([GPBlock(seq) | blocks], actorKs) -> [res1 | res2]
where
  res1 := <create-environments> (seq, actorKs)
;   res2 := <create-environments> (blocks, actorKs)
```

Figure 4.4: Stratego rule for choice interaction environment creation

sender, the rule fails with an error message. If it is found, the invariant information is added to the knowledge (K) of the appropriate actors using the `add-to-Ks` rule. The result of this rule is thus a set of updated actor knowledge, guaranteed to not contain duplicate invariant names or references to invariants unknown to an actor. On top of that, since `add-to-Ks` adds information to the knowledge of all actors involved in the message interaction, including receivers, receiving actors can now refer to the invariants introduced to them as well.

In summary, the rules used for the environment generation for message interactions ensure that all involved actors have a state added to their respective environments, and that their knowledge is updated with any newly received information. Due to the importance of this invariant knowledge, these rules became rather intricate. Since other interactions can neither introduce nor reference invariants, their rules are much shorter, focused on the structure of the environment rather than its contents.

Choice Interactions

While choice interactions do not involve invariants, their effect on an environment is prominent. Choices introduce a number of options which the program may choose. Each option is represented by a block containing more interactions. These options are contained in a single

```
1  create-environments: ([GIRecursion(_,block) | ints], actorKs) ->
2      [blockRes | intRes]
3  where
4    intKs    := <create-intKs> (actorKs, [block])
5  ; blockRes := <create-environments> (block,intKs)
6  ; intRest  := <create-environments> (ints,actorKs)
7
8  create-environments: ([GIContinue(_)  | ints], actorKs) -> [actorKs | res]
9  with (<?[]> ints < res := []
10                  + debug(!"interaction after continue statement");fail)
```

Figure 4.5: Stratego rule for recursion and continue interaction environment creation

list and represented by lists themselves, so that the states introduced by interactions within the blocks can be added to their respective lists. Processing the different options' interaction blocks is initiated using the same current set of actor knowledge, ensuring that any invariants introduced in one block are not known to a block following it. This is important, since only one block can be executed, so only invariants introduced before the choice interaction or within the current block can be used. This enforces the scoping restriction mentioned in section 4.1. Figure 4.4 shows the rule for handling choices. The result consists of three parts.

First, a choice interaction introduces its own state, created using the same `create-intKs` rule as message interactions. In this case, the rule returns the knowledge sets of all actors represented in any of the choice `blocks`, and thus taking part in the choice. This set is added to the environment to represent a new state.

The last result, `intRes`, is familiar from other interactions as well: it is simply the result of processing the interactions following the current one. Since a choice interaction does not affect actor knowledge, the current knowledge set `actorKs` can be used. Note that this also enforces restriction 4 from section 4.1: any invariant declared within the scope of the choice blocks is unknown to the interactions following them.

The result in between, `blockRes`, and its place in the overall result, are unfamiliar and carry consequences regarding scope. Applying `create-environments` to a list of blocks results in a list of those blocks' environments, each of which is a nested list. This rule, shown on line 8 uses the same set of actor knowledge for each block, ensuring the scoping mentioned before.

A choice interaction thus has no effect on actors' knowledge, but introduces a new nested list to the environment, containing the state information for the interactions in its choice blocks.

Recursion and Continue Interactions

A recursion interaction is modeled as a pointer to the pre-state of the first function in its block and, as such, introduces no new state of its own. Its enclosed block of interactions resembles the choice construct. It should therefore not come as a surprise that the environment generation rule shown in figure 4.5, line 1 has the same structure as the choice rule, except for not introducing a new state. It gathers a new set of actor knowledge based on the actors present in its recursion block, and uses only that set of knowledge to process the interactions within. The remaining interactions are processed with the current actor knowledge, ensuring the scoping rules hold.

A continue interaction simply creates a state using the current actor knowledge. Other than that, it only checks whether any interactions follow it, which should not occur. If it does, the code generation terminates with an error message.

4.1.2 Environment Generation for Running Example

This section will show the rules explained in the previous sections applied to the running example. Looking back at Figure 4.1(a) the global protocol of the interactions between Customer `C`, Seller `S`, and Banker `B` is shown. The resulting environments for each actor are shown in Figures 4.1(b) to 4.1(d).

Line 2 shows a recursion declaration. Section 4.1.1 explained that recursions do not induce a new state, but only a list containing the state information for the interactions with its interaction block. This list can be seen spanning between lines 2 and 14.

The next interaction is a message, containing an unnamed `String`. First, a state is added with the current actor knowledge for the actors involved in the interaction, in this case the empty lists for actors `C` and `S` on line 3. Since no new invariant information was introduced, actor knowledge is not updated. The state generated for `priceInfo` is an empty state as well. Here, `p` is introduced, so while the state generated for `priceInfo` is empty, the invariant information for `p` is passed as actor knowledge for actors `C` and `S`.

The choice interaction on line 5 introduces its own pre-state, as discussed in section 4.1.1. For actors `C` and `S`, this information contains `p`, as can be seen in their environments in Figures 4.1(b) and 4.1(c). The environment for `B`, however, contains only the empty list. A state is introduced for the choice interaction because `B` takes part in the choice options, but it has never been introduced to `p`, so the list of invariants representing the state is empty. Furthermore, a new list is started on this line, 5. This list encompasses all choice options, closing on line 13. This is the block containing the choice options discussed in section 4.1.1.

Lines 6 to 10 encompass the states for the interactions within the first choice block. The two `dummy` methods introduce states with the knowledge of the involved actors, and the continue statement introduces a state for all actors, since this interaction is relevant to all actors within the recursion block.

Lines 10 to 13 cover the states for the interactions within the second choice option. On line 11, `B` is involved in an interaction referencing `p`, but has never been introduced to it. According to the rules in figure 4.2, the sender, `C`, is searched for information regarding `p`. Since `C` was introduced to `p` and thus has its type information, `B`'s knowledge is updated with `p`. This shows in the state transition added for `ackTransfer` on line 12.

Since no interactions follow the choice or recursion interactions, the lists close and the environment generation is finished.

4.2 Idris Syntax Subset

Idris' syntax is not available in SDF format. Instead of trying to recreate it completely, a smaller syntax specifically for this project was designed that serves as an intermediate language, before pretty-printing Idris code with it. Figure 4.6 shows a simplified version of this syntax[2]. It serves as a target for the code generation based on a Scribble protocol and ensures the correct Idris keywords are used. Taking this responsibility away from the code generator itself reduces both the code size and possibility for errors. SDF's automatic pretty printing functionality allows for the use of proper indentation for the whitespace-sensitive Idris language.

As mentioned before, an Idris ST program consists of three parts: interface, protocol, and implementation. The syntax for a `program`, as seen in Figure 4.6(a), reflects this structure. Each of these parts is generated separately from the same Scribble protocol.

The `Interface` syntax shown in Figure 4.6(b) shows that each interface consists of a set of data declarations, such as the data declaration for the states; an interface declaration, in which the name and parameters of the interface are listed; and, representing the root of the

[2]The full syntax has more variations for pretty-printing specific state transitions, function structures, etc.

```
Program.Program =
<<Interface>
 <Protocol>
 <Implementation>
```

(a) General Syntax

```
Interface.IFace =
 <{DataDecl "\n"}+>
 interface <IFaceDecl> where
  <IBlock>>

IFaceDecl.IFaceDecl = <<IName> (<Name>)>
IBlock.IBlock = <<{FunctionDecl "\n\n"}+>>

FunctionDecl.FunctionDecl  = <<FunctionName> : <FunctionState>>
FunctionDecl.PFunctionDecl = <<FunctionName> : <{FunctionParam " -> "}+> -\>
    <FunctionState>>

FunctionState.MState = <ST <VarName> (<IReturnValue>) [<FunctionStateDecl>]>
FunctionState.MStatePH = <StatePlaceHolder>

FunctionStateDecl.MEmptyState  = <>
FunctionStateDecl.MStateChange = <<VarName> ::: <PreState> :-\> <PostState>>
FunctionStateDecl.MStateSame   = <<VarName> ::: <PreState>>
FunctionStateDecl.MStateAdd    = <add (<DataName> (<DataElement>))>
FunctionStateDecl.MStateAddIfJust = <addIfJust (<DataName> (<DataElement>))>
FunctionStateDecl.MStateRemove = <remove <VarName> (<DataName> <DataElement>)>

PreState.PreState    = <<DataName> (<DataElement>)>
PostState.PostState  = <<DataName> (<DataElement>)>
PostState.CPostState = <\\res =\> <DataName>
    (case res of <{StateCaseDecl "\n"}+>)>

StateCaseDecl.CaseNothing = <Nothing =\> (<DataElement>)>
StateCaseDecl.CaseJust    = <Just <TypeName> =\> (<DataElement>)>
StateCaseDecl.CaseJustAny = <Just _ =\> (<DataElement>)>
StateCaseDecl.CasePure    = <<TypeName> =\> (<DataElement>)>
```

(b) Interface Syntax

```
Protocol.Protocol = <using (<{IFaceRef ", "{+>)
  <{FunctionAny "\n\n"}*>>

IFaceRef.IFaceRef = <<IName> <VarName>>

FunctionAny.AFunctionImpl = <<FunctionImpl>>
FunctionAny.AFunctionDecl = <<FunctionDecl>>

FunctionImpl.FunctionImpl = <<FunctionName> <{Var " "}*> = <FunctionBlock>>
FunctionBlock.FunctionBlock = <<{FunctionLine "\n"}+>>
FunctionLine.Assignment   = <<Var> \<- <FunctionCall>>
FunctionLine.LineDo       = <do {<FunctionBlock>};>
FunctionLine.JustNothing  = <<JustNothing>>
FunctionLine.LFunction    = <<FunctionCall>>
FunctionLine.Hole         = <?<FunctionName>>
```

(c) Protocol Syntax

Figure 4.6: Simplified intermediary syntax used as target syntax for translation from Scribble to Idris

interface structure, an interaction block: `IBlock`. The block consists of function declarations representing the interactions, defining a name, optional parameters, and a state transition. Lines 15 to 20 represent the different forms a state transition can take. Here, the `<DataName> (<DataElement>)` pair represents the state, e.g. `StateT (C1)`. The remaining rules elaborate on the different forms of pre- and post-states. With these building blocks, the required function definitions can be described with the proper state transitions. How these are generated is explained in section 4.3.

A protocol consists mainly of functions with `do`-blocks. Figure 4.6(c) shows that a protocol consists of references to used interfaces followed by a list of functions. Unlike an interface, a protocol may contain both function declarations and implementations. The declarations have the same form as those in the interface, as seen in Figure 4.6(b) on line 9. The implementation form is shown in Figure 4.6(c) on line 9. It consists of a name, optional parameters, and the block defining the function's body. Lines 11 to 15 show the possible types of calls that can be made within a function's implementation. Section 4.4 elaborates on how this is used to generate the protocol for Idris.

The intermediary syntax for the implementation is not shown here, since the choice was made not to generate the implementation due to time constraints.

It should be made clear that this syntax is not representative of the full Idris syntax, mainly in that this syntax is very limited in its applicability and possibilities: it is solely used to generate code required for communication protocols in Idris. More importantly, it is ambiguous and allows constructions that would not be valid in Idris. These limitations are accepted since this syntax is not intended to parse Idris code, but only to print it. It is meant to only be used in one direction, with full control over the syntactical structures created with it.

The following sections elaborate on how this syntax is used to generate the Idris code for each of the ST program parts: interface, protocol, and implementation.

4.3 Interface

This section will describe how an Idris interface is generated from a Scribble protocol. Section 2.2.2 explained how a protocol's state transitions are defined in the Idris interface, as well as the input and output parameters for functions. These function signatures provide the structure for all the code in the protocol and implementation parts of the program and it is therefore critical that it is correct with respect to the original protocol.

Since the interface only contains function signatures, its main body consists of a single flat list. As such, generating an intermediary AST from Scribble means that for each visited interaction, its signature should be generated and simply added to the list. The main challenge for the interface, therefore, is not maintaining the Scribble AST structure, but rather assigning the correct parameters and states to the signatures. For this, the previously constructed environment is used.

```
/* ... */
rec r {
  reqPrice(String) to S;
  priceInfo(p : Integer) from S;
  choice at C
  {
    dummy() to B;
    continue r;
  } or {
    reqTransfer(p) to B;
  }
}
/* ... */
```

(a) Scribble Protocol for Customer

```
[
 [
  ("Ready",[]),
  ("C1",[]),
  ("C2",[(p,Integer)]),
  [
   ("C3",[(p,Integer)]),
   ("C4",[(p,Integer)])
  ],[
   ("C5",[(p,Integer)])
  ]
 ]
]
```

(b) Environment for Customer

```
{- ... -}
data State = Ready | C1 | C2 Int | C3 Int | C4 Int | C5 Int
data Choice = Option1 | Option2
interface Customer (m : Type -> Type) where
  {- ... -}
  reqPrice : (l : Var) -> (s : String) ->
    ST m () [l ::: StateT Ready :-> StateT C1]

  priceInfo : (l : Var) ->
    ST m Int [l ::: StateT C1 :->
      \res => StateT (case res of
        p => (C2 p)]

  choice : (l : Var) ->
    ST m Choice [l ::: StateT (C2 p) :-> \res => StateT
    (case res of
      Option1 => StateT (C3 p)
      Option2 => StateT (C5 p))]
  {- ... -}
```

(c) Translated to Idris

Figure 4.7: The origin, environment, and Idris result of a protocol translation

Each of the previously mentioned types of interactions (message, choice, recursion, and continue) result in different types of functions in the Idris interface. This section is split up into separate sections for each of them.

4.3.1 Message Interactions

Message interactions are the most basic interactions in a session type protocol. Section 2.2.2 explained how they transition from one state to the next, possibly with the introduction of an invariant. A Scribble message interaction has three attributes: its direction (sending/receiving), its name, and its payload. The name is used as the name for the new Idris function, and the payload influences either the function's parameters or return type, depending on its direction.

There are four types of payload:

- Empty payload `()`
- Base type `T`, where `T` is a valid type
- Invariant introduction `x : T`, where `x` is the name and `T` the type of the new invariant
- Invariant reference `x`, where `x` is the name of a known invariant

When a new invariant is introduced by a message interaction, that interaction's post-state and the states following it are parameterized by that new invariant's value. Where this value is defined depends on the direction of the interaction: either sending or receiving. This, and how the other payloads are handled, is discussed in the following sections dedicated to the directions.

Sending

As a message interaction, a sending interaction transfers only from one state to the next. The type of payload affects the parameters for the Idris function to be constructed, and the parameters of its post-state. A sending interaction has the following general signature structure in Idris, where the color boxes are placeholders for generated code:

```
ID : (l : Var) -> PARAMETERS ->
    ST m () [l ::: StateT PRE-STATE :-> StateT POST-STATE]
```

Here, `ID` is the name of the original interaction. The `PARAMETERS` are created based on the payload type(s) in the message:

- Empty payload `()` creates no parameter.
- Generic type `T` creates an unnamed parameter `T`.
- Invariant introduction `x : T` creates a named parameter `(x : T)`, where `x` equals a new parameter in the `POST-STATE` with the same name.
- Invariant reference `x` creates a dependent pair `(x' : T ** x' = x)`, where `x` is a parameter of type `T` of `PRE-STATE`.

The pre- and post-states are those states in the previously generated environment associated with this interaction. Section 4.1 explained how these states contain the invariant information they are parameterized by. The Stratego code used to perform this transformation can be found in appendix B.1.

Receiving

As opposed to sending interactions, receiving interactions only have one function parameter. Their return type and state transition, however, do depend on the expected payload type. If no new invariant declaration is defined in the payload, the return type and its effect on the state transition is very similar to that of a sending interaction, the main difference being that the function's return type is a tuple consisting of the payload types, rather than the list of parameters.

```
ID : (l : Var) -> ST m RETURN [l ::: StateT PRE-STATE :-> StateT POST-STATE]
```

When a new invariant is declared in the payload, however, the `POST-STATE` requires a value for its new invariant parameter, which is now enclosed in a variable in the tuple that is the return type. This variable is accessed using a `case` statement, pattern matching on the pattern of the `RETURN` tuple:

```
ID : (l : Var) -> ST m RETURN [l ::: StateT PRE-STATE :-> \res => StateT
  (case res of
    PATTERN => POST-STATE)]
```

An example of such a function is `multipleReturns`, which has an unnamed `String`, known invariant `i` of type `Integer`, and new invariant `b` of type `Bool` as a payload:

```
multipleReturns : (l : Var) -> ST m (String, (i' : Integer ** i' = i), Bool)
  [l ::: StateT (T1 i) :-> \res => StateT (
    case res of
      (_,_,b) => (T2 i b))]
```

The known invariant `i` is represented by a dependent tuple, ensuring its value is equal to that of state `T1`'s parameter. The new invariant is identified in the pattern of the `case` statement. The value of the other return types is irrelevant for his pattern match.

The Stratego code used to perform this transformation can be found in appendix B.2.

4.3.2 Choice

A choice interaction allows a protocol to choose a course of action based on a decision made by one of the actors. For the decision-maker, the logic of the decision is programmed in the function, providing the result as a return value after sending it to the other involved actors. For the receivers of the decision, the result of the decision is received from the decision-maker and then returned. This means that, although the implementations are very different, the signatures generated by the tool are actually the same:

```
choice_ID : (l : Var) -> ST m Choice_ choiceID [l ::: StateT PRE-STATE :-> \res =>
    StateT
  (case res of
    choiceID1 => POST-STATE1
    choiceID2 => POST-STATE2
    BRANCHES)]
```

Here, `ID` is a unique identification for each choice in the protocol. The return type is a data type generated for each choice, consisting of a number of `choiceID`s equal to the amount of options available to this choice. The `POST-STATE`s are dependent on the result and are gathered from the environment. They represent the pre-states of the first interaction in the interaction block associated with the resulting decision.

The Stratego code used to perform this transformation can be found in appendix B.3.

4.3.3 Recursion

Recursion and continue statements are used to indicate where a recursion block starts and ends. Implementing recursion is done by creating a separate function in the protocol section, as described in section 4.4. As such, they do not create a function signature in the interface. They do, however, indicate at which state the recursion starts. This is important for the continue statement. Continue statements are used to indicate when a recursion block is to be repeated. Since they serve no other purpose, they do not have their own functions. Instead, when a continue interaction is encountered, the previously mentioned starting state of the recursion is used as a post-state for the interaction right before the continue statement. This way, the program transitions to the correct state before repeating the recursion.

4.4 Protocol

With the interface specified, the protocol can be created. Before the protocol generation is started, some preparations are made. The required interface references are created, by default `ConsoleIO` and, of course, the interface that was just generated. Then, the type of the `protocol` function is defined. This is always the same:

```
1 protocol : ST io () []
```

The protocol function returns nothing, and starts and ends statelessly, since it creates and terminates the stateholder within its implementation. `io` is, by convention, the name the protocol section uses for the underlying context. The implementation consists of a set of calls listed in the Idris `do`-notation. This list of interactions represents the protocol as defined in the Scribble protocol, although the exact notation of both languages are different.

As with the interface, the interactions are the most important artifacts to translate and are thus discussed in this section. First, the message interactions are elaborated upon. The same cases as the interface will be encountered for the sending and receiving interactions. The next interaction to be explained is the choice interaction. Its different choice branches lead to a branching in the Idris protocol as well. Then recursion is discussed. Recursion has a special place in the protocol section, as will be explained.

4.4.1 Message Interactions

The payload that messages can carry affected is not only the signatures, but also the states in the interface. This section will show that the different possibilities of payloads discussed in section 4.3.1 are the same as what the protocol generation deals with. First, the calls generated for sending interactions are discussed, followed by those for the receiving interactions.

Sending

A sending interaction is a simple call, the structure of which depends on its payload. When a new invariant is introduced by the interaction, a line is added to introduce it in the program using a `let` statement. The general structure is shown here, where the cyan box indicates the optional code, and the yellow boxes generated code.

```
1 let x = DEFAULT
2 ID PARAMS
```

The `ID` and `PARAMS` are the names of the interaction and the payload's parameter names. The cyan `let` statement is added when the sending interaction introduces a new invariant. This invariant will be referenced in `PARAMS` and therefore must have a value assigned to it. For each new invariant declared within the payload, one of the `let` lines is generated. The value `DEFAULT` is a hard-coded default value for the type of `x`.

Hard-coding the values of new invariants is far from ideal, but the Idris type checker requires a value for the program to type check. For now, hard-coding is a quick way to achieve a valid program and automatically type check them. Other options exist, such as asking the user for a value or obtaining them from an external source, but these require additional time either when testing or when generating code.

An example of a let binding and sending interaction can be found below in 4.8(b), lines 12 and 13. The Stratego code for performing the transformation for the case of invariants being present in the payload can be found in appendix C.2.

Receiving

A receiving interaction is also a relatively simple call, having the same `ID` and `PARAMS` as a sending call:

```
1 x <- ID PARAMS
```

Here, however, there is only one variable assignment. As described in the previous section, the return type of a receiving interaction is a tuple when more than one return value is defined. In those cases, `x` is also a tuple, consisting of the parameter names of the expected return values, so that they can be used later in the protocol. The Stratego rules that perform this translation can be found in appendix C.3.

4.4.2 Choice

Choice interactions result in the execution of different interactions based on the choice made. Section 4.3.2 explained that a choice interaction returns a value of a dedicated data type that determines the course of action. In the protocol, a `case` match is used to make this distinction:

```
1 decision <- choice_ID l
2 case decision of
3   OPTION1 => BLOCK
4   OPTION2 => BLOCK
5   {- ... -}
```

Depending on the value of `decision`, a certain `BLOCK` interaction block is executed. This block is a set of interactions defined in the `do`-notation.

An example of a choice interaction in a protocol can be seen below in Figure 4.8(b). Line 14 shows the choice call, with the next line starting the `case` analysis. The exact Stratego rules can be found in appendix C.4.

4.4.3 Recursion

Recursion requires the possibility to make a call to the starting point of the recursion. This is modeled in Idris by creating a separate function for a recursion block, which requires both a signature and implementation. The implementation consists of generating the Idris code for the interactions within the recursion block, generally as with a normal protocol function. The signature of the function is a simple state transition. The choice was made to keep any invariants declared within the recursion block within that block's scope[3] and, as such, the recursion function has no return value. The signature is therefore of the form:

```
1 recursion_ID : (l : Var) -> ST io ()
2           [l ::: StateT {m=io} PRE-STATE :-> StateT {m=io} POST-STATE ]
```

Here, `ID` is the label of the recursion variable declared in the Scribble protocol, and the `PRE-STATE` is the state in which the recursion starts. The `POST-STATE` is the state in which the recursion should be when it ends. It is therefore the pre-state of the interaction directly following the recursion block.

In the example in figure 4.8, the original protocol ends after the recursion is finished. As such, the post-state of the recursion function is the final `Done` state.

The implementation of the recursion function is similar to the protocol implementation, only differing in what happens at the end of the block. The implementation therefore looks like the following:

```
1 recursion_ID = do BLOCK
```

[3] All invariants declared before the recursion are available within the block

Here, `ID` is the same as the signature, and the `BLOCK` is generated using the normal protocol code generation rules. Figure 4.8(b) shows an example of the `recursion_r` function on line 10.

Of course, a recursion should be able to recurse. In Scribble, this is declared using the `continue` statement. Section 4.3.3 explained that the interaction right before this statement has as a post-state the state in which the recursion starts. Therefore, it is possible to directly call the recursion function from the place the `continue` is encountered. Line 18 in figure 4.8 shows such a call.

The code for creating the call to the recursion function, the function signature, and its implementation can be found in appendix C.5.

```
1  /* ... */
2  local protocol Shopping_S (role C, role S, role B){
3    rec r {
4      reqPrice() from C;
5      priceInfo(p : Integer) to C;
6      choice at C {
7        dummy2() from B;
8        continue r;
9      } or {
10       ackTransfer(p) from B;
11 } } }
```

(a) Scribble protocol

```
1  {- ... -}
2    shopping_S : (l : Var) -> ST io () []
3
4    recursion_r : (l : Var) -> ST io () [l ::: StateT {m=io} State1 :-> StateT {m=io}
       Done]
5
6    shopping_S l =
7      do recursion_r  l
8         done_Shopping_S  l
9
10   recursion_r l =
11     do reqPrice  l
12        let p = 0
13        priceInfo  l p
14        choiceResult <- choice1  l
15        case choiceResult of
16         Option1 =>
17           do dummy2  l
18              recursion_r  l
19         Option2 =>
20           do ackTransfer  l
21              pure()
```

(b) Resulting Idris protocol

Figure 4.8: Example protocol code generation

4.5 Implementation

The implementation has two important responsibilities: one is implementing the logic for e.g. choices, invariant values, etc. Another is handling the actual communication between parties.

The first responsibility is based on information that is not currently available in Scribble. The choices, for example, only specify which actor makes the decision, not how that decision is made. Considerations on how that may be improved are discussed in chapter 7. Until then, it is up to the Idris programmers to implement this last piece of information.

The second part, handling the communication, consists of default, boiler-platey code that should be possible to generate. Exact details such as addresses are, of course, unknown, but the composite resources seen in chapter 2, used for the sending and receiving of messages, seem like a possible target for code generation.

For now, however, no implementation is created at all.

4.6 Conclusion

After this chapter, the reader should be familiar with how Idris code is generated from a protocol specified in Scribble. The syntax to which a protocol is translated is first introduced, so that readers can follow the rules in later sections. Keeping track of states and invariant scope is crucial to the proper functioning of the protocol as a whole. The reader is now familiar with how this is recorded and used. Finally, the transformation from ScribbleI to Idris is shown. Chapter 5 discusses how this code generation is tested, as well as assessing how the described tool meets the thesis research objective.

Chapter 5

Evaluation

The previous chapters described the tools created to provide support for implementing communication protocols with invariants. This section is devoted to evaluating those tools. Section 5.1 illustrates the tests performed to evaluate the correctness of the Idris code generation. The usefulness of the contributions in a more general sense is discussed in section 5.2.

5.1 Testing the Idris Code Generation

The previous sections explained the rules that the code generation tool adheres to in order to create correct Idris code. Testing the results is an important part of this project. Testing was an integral part of development. During all stages any additions or modifications of rules was tested against a small test set to ensure their correctness. Depending on the work at hand, these tests consisted of protocols with particular interactions, invariants, or combinations thereof. For the early stages, where no Idris code was yet output, the results were checked manually against their expected output. Later, when the generated Idris code was expected to be valid, type checking it in Idris became the main verification method of modifications to the generator.

This section discusses the testing methods in-depth. In section 5.1.1 the testing method and its challenges are discussed. Section 5.1.2 discusses what test cases were used. The results of these tests is discussed in section 5.1.3.

5.1.1 Testing

Testing the code generated by the Stratego rules introduced a number of challenges. This section discusses these challenges and their solutions, ending with a summary of the testing methods.

The generator is built to generate an Idris program based on the logic encoded in a Scribble protocol. For the generation itself, translating the code without too much regard of the underlying logic suffices. Testing the result, however, should include testing whether the intended logic is represented in the Idris program. Since no implementation is created yet, executing the Idris program and checking its results is not an option. Instead of creating a test suite that could interpret this logic, testing the logic was done by manual inspection. This was deemed sufficient due to Idris' strict typing system. each Idris program should, of course, type check. If it does not type check, it is due to one of the following errors:

- Syntactic or semantic error
- State transition error
  - Incorrect state transition definition

- Missing function call
- Incorrect invariant

General syntactic or semantic errors are considered those that violate syntax or simple typing rules. Their cause is *structural*, in the sense that they are often found in the hard-coded parts of the translation tool, such as a missing required code structure in the intermediary translation syntax. State transition errors are more to do with the logic of a protocol. The state transitions represent the order of execution of functions. If one of them is incorrect in the interface or the protocol definition, the odds of that program passing the Idris type checker are deemed low.

Another challenge in testing is checking whether an *incorrect* protocol is rejected for the proper reason. Logic critical to the proper functioning of the code generator is usually wrapped in a `with` clause in Stratego. This ensures the program terminates, but also means that no result is returned. A debug message is therefore printed, indicating the error encountered. In case of a failed execution, this last message can be manually checked to correlate to the expected failure point.

With these challenges in mind, testing was done manually for test cases supposed to fail and cases testing whether Scribble's logic was retained. Test cases focusing on technical aspects of the translation were checked by executing the code generation and automatically type checking the results with Idris. What these test cases entail is discussed in the next section.

5.1.2 Test cases

Three sets of test cases were used to test the generator and its results:

- Specific test cases
- Example scenarios
- Scribble demos

The first set consists of small protocols designed to test a specific behavior of the generator. Appendices D.1 to D.4 list these for choices, recursion, payload types and scope. These tests were run by generating the Idris code and type checking it.

Example scenarios are bigger tests, useful for testing both the retaining of logic and the combined use of all interactions. Appendix D.5.1 shows the protocol for an ATM interaction by a user. This protocol was chosen due to its use in session type literature (Bonelli and Compagnoni 2007; Dezani-Ciancaglini and Liguoro 2009; Bocchi et al. 2013). Appendices D.5.2 and D.5.3 show (adapted versions of) the SEPA Credit Transfer and Direct Debit protocols[1]. These protocols are issued by the European Payments Council and used by many major banks. Finally, the running example is also tested, the Scribble protocol of which can be found in appendix E.1.

The Scribble demos refer to a set of Scribble demos available on the Scribble Github[2]. These demos cover a wide array of scenarios, including an HTTP protocol, a negotiation, and a loan request. The advantage of using these demos is that they provide a number of protocols tested for their functionality and making use of the wide range of Scribble potential. This allows not only the testing of the Idris code generation, but also that of the Scribble protocol transformation tools discussed in 3.1. The latter is also the downside: since this report covers a subset of Scribble syntax, some interactions in the demos are not available for

[1] European Payments Council 2017a; European Payments Council 2017b.

[2] Scribble on Github (n.d.). URL: `https://github.com/scribble`.

us. In order to be able to test these demos, they had to be altered to comply with the covered Scribble subset. This meant removing:

- Type declarations
- `do` interactions
- `connect` interactions
- Some keywords of protocol declarations (e.g. `aux`)

Also, payloads were either converted to unnamed types accepted by our syntax, or converted into invariant declarations. After these alterations, Idris code could be generated from them.

Notes and observations on these demos, along with the results of the other test cases are discussed in the next section.

5.1.3 Results

Almost all test cases passed, and all shown scenarios passed the test for code generation, Idris type checking, and retaining their logic. Still, some tests should be taken a look at. First, some failing specific test cases are considered. Then the results of the Scribble demos are discussed. Since the test scenarios are passed, no more mention of them is made.

There are two classes of failed test cases.

One has to do with introducing a variable known to the sender, but unknown to the receiver in the last interaction of a block. In case `InvariantIntroducedAtEnd` on line 41 in appendix D.4, actor `B` introduces invariant `i` to actor `C`. `i` has been previously introduced to `B`, but is unknown to `C`. Due to the design decision to only model pre-states in the environment, the information of `i` is lost to `C` in the local protocol, and an error occurs. A refactoring of the environment generation could eliminate this bug. Until then, it can be solved by adding a `dummy()` interaction after the violating interaction.

The second set of failed test cases has to do with empty blocks, either `choice` or `continue` statements. An example is `NoMessage` on line 26 in appendix D.2: Recursion Tests. Here, a recursion without a message interaction is defined. For now, a message is expected within a block.

These two cases are bugs in the current implementation. When testing the Scribble demos, on the other hand, it became clear that Scribble is indeed more expressive than ScribbleI. Several demos maintained their logic even after modification, but others lost it. A clear example is the `game.scr` demo. This demo depends much on delegating sessions to other protocols using the `do` interaction. While a design retaining its logic using only our syntax may be possible, it would mean much duplicated code and decreased readability.

Similarly, the `connect` interaction, which is used to connect two actors with each other, allows for the late introduction of an actor to the conversation. In the running example, `dummy` messages are required to handle the Bank's involvement in the `choice`, even if the Bank is actually only supposed to interact in one branch. Not connecting the Bank until it is required would resolve this issue.

In conclusion, the code generation works for the majority of cases it is currently designed for. Some edge cases not solved yet have been described. However, extending the syntax would allow for more complex protocols.

5.2 Contributions

This project presents three contributions: a method for implementing session types with invariants in Idris, ScribbleX, and a translation from ScribbleX to Idris. This toolset was created

to enable protocol developers to design safe protocols with invariants in a developer-friendly language and generate a working program that is guaranteed to follow that protocol's specification. This section will discuss whether the results match the initial intentions.

Idris The first objective was to explore the possibilities of creating an executable program that implements a communication protocol with invariants. Idris was chosen due to its dependent types, which can be used to statically ensure a value remains invariant, and its ST Library, which can be used to implement session types. This combination allowed for the desired implementation, where invariants in a communication protocol are indeed enforced. What is lacking, however, is an explicit notion of whether the implemented protocol adheres to *session type* protocol guarantees. These guarantees (linearity, progress, session fidelity, and communication safety) are the main strength of session types, and being able to prove they hold true for an implementation would be useful. Based on the experience gained during this project, such a proof may be possible. In Honda, Yoshida, and Carbone (2008), the proofs of the latter three guarantees are reliant on the linearity property. As mentioned in section 6.1, a protocol is linear when no message can be mistaken for another. In session types, this is inherent in the typing system. In an implementation techniques such as message queues may lead to the same result. Assuming that this property holds, the other three can be reasoned to hold as well. Progress is a consequence of the structure of the protocol, which is enforced by Idris' state transition system. This is also true for session fidelity: it is impossible for an actor to deviate from the protocol. Communication safety is a little more complicated with the introduction of invariants. If all actors are generated from the same global protocol, the environment generation should ensure that all invariant knowledge is correct for each actor. As such, no received message content should be unexpected or incorrect, and this property should hold as well. Therefore, generating a program from a valid Scribble protocol increases the likelihood of these properties to transfer to an Idris protocol, but we have not provided proof to establish a significant claim.

ScribbleI The second objective was to enable relative laymen in programming (with dependent types) to easily design protocols and generate a machine executable program from them. For this reason, Scribble was chosen as a base language. Scribble's small and straightforward syntax allows for rapid design and prototyping, and its existing algorithms to check for session type adherence ensure the safety of protocols. Extending its syntax to allow the specification of invariants was a small, but effective modification. Checking the session type safety of a ScribbleI protocol is possible, albeit in a roundabout manner. The small tool that creates a Scribble protocol from a ScribbleI protocol allows the user to use the original Scribble checker. This was purposely done to be able to rely on the expertise of its creators. The proper use of invariant declarations and references is also checked and tested, but only with respect to the actor knowledge and scope. Intuitively, it seems these invariants should not interfere with session type safety, but we provide no proof regarding this notion.

Translation The Idris code generation from a ScribbleI protocol saves time and reduces the chance of human-induced errors in the implementation. The test results show that the generator adheres to the protocol definition for all actors and creates an interface and protocol that retain the original protocol's logic. The state transition system used to enforce message ordering ensures that changes that would invalidate this ordering result in a type error. However, the lack of an implementation generator means that knowledge of Idris is still very much required. This lack is due to time constraints and it is our belief that such a generator can be created, so as to minimize the need for manual completion of the generated Idris program. *Minimize*, because, for now, ScribbleI (and Scribble) lack the ability to express certain logic required in an implementation, such as what the decision logic in a choice inter-

action. Furthermore, a formal proof of the correctness of the code generator would solidify its usefulness, especially with respect to maintaining the session type guarantees.

Chapter 6

Related Work

This research is built upon a number of different research fields. This section will discuss this relation with respect to relevant existing research.

First, since this work is heavily inspired by them, the multiparty session type theory by Honda, Yoshida, and Carbone (2008) will be explained in more detail. Then, other projects involving adding dependent types and/or invariants to session types are discussed. Finally, Idris and the ST library are discussed.

6.1 Multiparty Asynchronous Session Types

Session types were introduced by Honda, Vasconcelos, and Kubo (1998) as a way to formally describe communication protocols by typing them. This first iteration used *binary* protocols: exactly two actors were involved in the protocol. Since then, session types have been actively researched and improved. Dezani-Ciancaglini and Liguoro (2009) provide a comprehensive overview of session types and their growth. A major milestone was fully functional multiparty asynchronous session types by Honda, Yoshida, and Carbone (2008). This theory is explained in more detail in this section.

Communication protocols are formal descriptions of communication between two or more parties. In session types, these protocols consist of a sequence of interactions, which represent a participant's (actor's) actions, such as sending a message or making a choice. Session types cover three levels of abstractions: global protocols, local protocols, and processes. Global protocols describe a communication protocol from an all-knowing perspective: all interactions of all actors are described in it. Local protocols are more specific protocols, which are described from the perspective of a single actor. This means, for example, that interactions in which the current actor is not involved are not present in its local protocol. Processes describe the behavior of the actors on a concrete level. The relation between these layers is formally defined: a global protocol is *projected* onto an actor to create a local protocol using a set of rules, and a *type system* ensures that a process adheres to this local protocol. The formal nature of session types allows for proofs regarding certain properties of such protocols. Multiparty asynchronous session types allow for communication with two or more actors in an asynchronous setting, and are proven to provide the following guarantees:

- Communication Safety
- Progress
- Session Fidelity

Communication Safety means that no communication errors will occur. Communication errors in this context are errors regarding the timing of message sending and reception: there is never a mismatch between the types of sent and expected messages, even if the same

Language	# Actors	Authors
Haskell	Binary	Neubauer and Thiemann (2004)
$\mathcal{L}_{doos}$	Binary	Dezani-Ciancaglini, Yoshida, et al. (2005)
MOOS	Binary	Dezani-Ciancaglini, Mostrous, et al. (2006)
MOOS$_<$	Binary	Dezani-Ciancaglini, Giachino, et al. (2006)
Java	Binary	Hu, Yoshida, and Honda (2008)
Java	Binary	Gay et al. (2010)
C	Multiparty	Ng, Yoshida, and Honda (2012)
Java (Scribble generated)	Multiparty	Yoshida, Hu, et al. (2013)
Java (Mungo/StMungo)	Multiparty	Kouzapas et al. (2016)
Scala	Binary	Scalas and Yoshida (2016)

Table 6.1: Implementations of Session Types

communication channel is used for exchanging messages of different types (Coppo et al. 2015).

Progress is the fact that every message sent is eventually received, and every actor waiting for a message eventually receives one.

Session fidelity means that, after a session has been started, no actor will deviate from their prescribed protocol. This is an important feature, considering that messages in session types have no identity: one message can not be directly distinguished from another.

Session fidelity is inherited from the binary session type theory, since it can be checked at a local level. The other two guarantees, however, require a global approach. A key property when proving them is that channels are used *linearly*. *Channels* are the communication channels across which actors communicate with each other and their use plays an important role in allowing actors to process interactions in the correct order. Channels are linear when no incoming message content can be mistaken for another, despite messages having no identity and an asynchronous environment. Honda, Yoshida, and Carbone (2008) provide an excellent example of how this is required using the two-buyer protocol. When channels are used linearly, both progress and communication safety can be shown to hold for a protocol.

Multiparty session types have two syntaxes: the calculus syntax, which is used to prove the previously mentioned guarantees, and that of the communication type system, which is used to check type soundness of the communications and protocol fidelity. Scribble's syntax is based on that of the communication type system, which it uses to check whether the previously mentioned guarantees hold.

6.2 Related Research

Session types have inspired a multitude of research directions since their inception in the late nineties. This section will provide an overview of research related to this project.

Session Type Implementations Session types have been implemented in multiple languages, with varying capabilities. Table 6.1 shows a summary of such implementations. Neubauer and Thiemann (2004) created a `Session` monad for Haskell. This adhered to all the type rules of session types, but only handles protocols with exactly two actors (binary). $\mathcal{L}_{doos}$ (Dezani-Ciancaglini, Yoshida, et al. 2005) and MOOS (Dezani-Ciancaglini, Mostrous, et al. 2006; Dezani-Ciancaglini, Giachino, et al. 2006) were two small object-oriented languages created with the intent to implement session types. This work served as a basis for Hu, Yoshida, and Honda (2008), who created the first Java implementation of session types by extending Java with session primitives and subtyping. They used a combination of static and dynamic checks to ensure the implementation adhered to session type properties. Gay et al.

(2010) also created an implementation for session types in Java, which provided a modularity not seen in previous work, allowing for more flexible use of channels by actors. These implementations are all for *binary* session types. This project provides an implementation for multiparty session types, next to the addition of invariants.

While previous work could mimic multiparty communication using session delegation, the first implementation of multiparty session types is by Ng, Yoshida, and Honda (2012), creating a toolchain for C in which global session types can be expressed, projected unto local types and finally resulting in endpoint protocols, based on which the resulting C program is designed. A similar design pattern resulted from Scribble Yoshida, Hu, et al. (2013), for which a Java generator was created in cooperation with JBoss[1]. The generated code represents the structure of communication, limiting the implementation to interactions specified by the protocol. In a similar vain, Kouzapas et al. (2016) created Mungo and StMungo. Mungo extends Java with typestate definitions, which enables a developer to assign states to classes. StMungo is a tool that generates a typestate definition based on a Scribble protocol. This project follows the same general structure: create a high-level protocol description and use a tool to generate an implementation for it. A novel contribution, however, is the addition of invariants on each level of abstraction, while maintaining session type guarantees on the global and local protocol levels.

Dependent Types and Session Types Dependent types have been used in combination with session types before. Yoshida, Deniélou, et al. (2010) use dependent types to model parameterized multiparty session types. This allows for the design of protocols with dynamic parameters, like the number of participants taking part. However, only the parameters of the protocol are modelled with dependent type, not the contents of message payloads. The limitations with respect to knowledge about the payload is therefore still lacking. Toninho, Caires, and Pfenning (2011) develop an interpretation of intuitionistic linear type theory as a dependent session type system, which not only types protocols, but also adds properties of message payloads. While similar, their work only supports binary session types. Toninho and Yoshida (2017) have defined a type discipline for session types with value dependent payload. Like our work, it is based on multiparty session type theory by Honda, Yoshida, and Carbone (2008) and treats values as singleton types. Moreover, dependent types are fully supported in their work, including their use as payload. To the best of our knowledge, however, this theory exists only on pen-and-paper for now, whereas this project provides a machine executable implementation of invariant payloads.

[1] JBoss (n.d.). URL: http://http://www.jboss.org/.

Chapter 7

Future Work

Chapter 5 discussed in how far the current contributions provide a solution to the initial problem statement. The previous chapter put them in context with existing work. This chapter will describe what could be done in the future to improve upon this work.

Correctness Proofs We provide a way to implement communication protocols in Idris, based on a session type protocol. The lack of a formal description of the code generation means that there is no proof that the implementation adheres to the original protocol. Tests show that this is usually the case, but it is our belief that Idris' formal typing system could be used to create a formal proof that supports this intuition.

Subprotocols and Parallelism Scribble allows for the specification and execution of subprotocols, allowing the design of modular protocols. This should be possible to define in Idris, but care should be taken when checking for invariant violations in a set of global protocols like these. The same is true for Scribble's explicit parallel execution capabilities. Toninho and Yoshida (2017) incorporate this in their theoretical framework, which suggests that this should be possible in an implementation as well. Even so, the notion of invariants in a parallel environment may introduce new challenges.

Improved Code Generation The current Idris code generator produces Idris type signatures as output, and it is up to the developer to develop a well-typed implementation of each function in the interface. Idris has support for programming with holes, that is, at any point during the development of a program, you can interactively inspect the state of the type context and see what the rest of the protocol expects in order to type check, which in our experience is a useful methodology for type-driven implementation of multi-party protocols. In future work, we would like to investigate how to improve the code generation along several dimensions:

- The safety of a protocol in general depends on which channels actors are communicating along. We expect it would be relatively straightforward to support (or perhaps even require) channel annotations for each interaction in our variant of Scribble, and to generate type signatures that guarantee that a given transition must send or receive a message along the channel.

- Our illustration of how to implement a protocol used the socket module that is a part of Idris' ST library. To model something like channels, it is desirable that we implement better library support for strongly-typed network programming, such as length-indexed message queues and/or type-indexed heterogeneous message queues.

- We have no generic method of parsing incoming messages. Creating a parser for basic session types might be trivial, since messages and types are expected. However, future work with expanded content capabilities will complicate parser design. Therefore, a more generic parser, made to handle less regulated communication, is desirable, possibly as part of the previously mentioned networking library.
- The current Idris code generator generates type signatures and a protocol with holes for application-specific logic, such as choosing what concrete value to pass from one actor to another. In future work, it would be interesting to see how far we can take the code generation approach: could we generate entire implementations of local protocols and have a dependently typed host language verify that the generated implementation is safe w.r.t. its specification?
- Our Idris type signatures currently only provide guarantees about *local* protocols, whereas the verification of properties about *global* protocols is delegated to the Scribble implementation itself. It seems attractive and tractable that we should be able to reason about the composition of a collection of local protocols, to verify in a dependently typed host language like Idris that the protocols satisfy global properties like linearity.

Shared Knowledge Management We briefly discussed how the simple customer-seller-bank protocol was subject to an invariant at a critical point at which all actors should agree on the state of the singleton type denoting the price of the product that funds were being exchanged for in the protocol. It is currently beyond both regular Scribble, ScribbleI, and other frameworks for dependent session types, such as Toninho and Yoshida (2017) and Wu and Xi (2017), to statically enforce that the knowledge of dependently typed invariants has been propagated correctly to all actors. As argued and illustrated in connection with the customer-seller-bank example protocol, it is useful do so. There are several potential strategies for ensuring that actors agree on invariants. One such strategy which is currently en vogue is block-chain technology where common knowledge is kept track of in a ledger. Less involved options exist, such as a classification of which actors in a given global protocol are credible sources to receive knowledge of a singleton type from. Another scheme would be to annotate data with an unforgeable crypto-signature, whereby an actor at a critical point in a protocol is able to identify and query the originator of a singleton type, to validate that the originator agrees with the singleton type state at that point in the protocol.

Chapter 8

Conclusion

This report has shown the progress made towards this project's objective to:

> *Support development of machine executable implementations for multi-party protocols with dependently typed invariants.*

First, Idris' potential for implementing session types using the ST Library was shown in chapter 2. Its interface can be used to enforce a fixed ordering of interactions through stateful signatures. These signatures were used to define a protocol in which the program was made more concrete. The implementation of the interface uses the Network library to connect a protocol's actors and provide a machine executable implementation of the whole protocol. Finally, invariants were added to this basis. By parameterizing states with known invariants, their invariance can be guaranteed for each actor individually.

While it is possible to create protocols from scratch in Idris, there is a number of downsides to this. It requires expertise to work with Idris, which means that either protocol designers need to learn this expertise, or a designer's protocol needs to be implemented by a programmer. This extra step could introduce discrepancies between a designer's intention and the final result due to the difficulties of explaining a protocol without a formal method. Therefore, chapter 3 introduced Scribble, a multi-party session type protocol language. This small, developer-friendly language enables a developer to quickly design a protocol and test it for session type safety. Scribble, however, does not support invariants. To extend Scribble with these, a subset of its syntax was implemented in Spoofax, named ScribbleI, along with checks to ensure correct use of invariants in a protocol. The projection of a global protocol to a local protocol was also implemented, as well as a tool that transforms a ScribbleI protocol to a Scribble protocol. This latter tool is used to check whether a ScribbleI protocol is still a valid session type.

Finally, a code generation tool was described, which generates Idris code from a ScribbleI protocol. The generated code represents an interface and protocol definition for Idris' ST Library. The tool was shown to be able to translate a variety of ScribbleI protocols to working Idris code, maintaining both the logic and restrictions of the original protocol.

This project showed that machine executable implementations of multi-party communication protocols with invariants is possible, using Idris' ST Library and dependent type checker to enforce protocol execution rules. Additionally, creating such implementations was made easier and less error-prone by generating the Idris code from ScribbleI. While improvements are desirable, the groundwork for full tool support for implementing safe protocols has been laid.

Bibliography

ANTLR (n.d.). URL: http://www.antlr.org/.

Bocchi, Laura et al. (2013). "Monitoring Networks through Multiparty Session Types". In: *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOOD-S/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*. Ed. by Dirk Beyer and Michele Boreale. Vol. 7892. Lecture Notes in Computer Science. Springer, pp. 50–65.

Bonelli, Eduardo and Adriana B. Compagnoni (2007). "Multipoint Session Types for a Distributed Calculus". In: *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*. Ed. by Gilles Barthe and Cédric Fournet. Vol. 4912. Lecture Notes in Computer Science. Springer, pp. 240–256.

Brady, Edwin (2013a). "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5, pp. 552–593.

— (2013b). "Idris: general purpose programming with dependent types". In: *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*. Ed. by Matthew Might et al. ACM, pp. 1–2.

— (2016). "State Machines All The Way Down: An Architecture for Dependently Typed Applications". Unpublished draft.

Coppo, Mario et al. (2015). "A Gentle Introduction to Multiparty Asynchronous Session Types". In: *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*. Ed. by Marco Bernardo and Einar Broch Johnsen. Vol. 9104. Lecture Notes in Computer Science. Springer, pp. 146–178.

Dezani-Ciancaglini, Mariangiola, Elena Giachino, et al. (2006). "Bounded Session Types for Object Oriented Languages". In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4709. Lecture Notes in Computer Science. Springer, pp. 207–245.

Dezani-Ciancaglini, Mariangiola and Ugo de Liguoro (2009). "Sessions and Session Types: An Overview". In: *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers*. Ed. by Cosimo Laneve and Jianwen Su. Vol. 6194. Lecture Notes in Computer Science. Springer, pp. 1–28.

Dezani-Ciancaglini, Mariangiola, Dimitris Mostrous, et al. (2006). "Session Types for Object-Oriented Languages". In: *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*. Ed. by Dave Thomas. Vol. 4067. Lecture Notes in Computer Science. Springer, pp. 328–352.

Dezani-Ciancaglini, Mariangiola, Nobuko Yoshida, et al. (2005). "A Distributed Object-Oriented Language with Session Types". In: *Trustworthy Global Computing, International Symposium,*

TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers. Ed. by Rocco De Nicola and Davide Sangiorgi. Vol. 3705. Lecture Notes in Computer Science. Springer, pp. 299–318.

European Payments Council (Nov. 2017a). *SEPA Credit Transfer Rulebook*. 11th ed. https://www.europeanpaymentscouncil.eu/document-library/rulebooks/sepa-credit-transfer-rulebook. European Payments Council.

— (Nov. 2017b). *SEPA Direct Debit Core Rulebook*. 11th ed. https://www.europeanpaymentscouncil.eu/document-library/rulebooks/sepa-direct-debit-core-rulebook-3. European Payments Council.

Gay, Simon J. et al. (2010). "Modular session types for distributed object-oriented programming". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, pp. 299–312.

Honda, Kohei, Vasco Thudichum Vasconcelos, and Makoto Kubo (1998). "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *Programming Languages and Systems - ESOP 98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Chris Hankin. Vol. 1381. Lecture Notes in Computer Science. Springer, pp. 122–138.

Honda, Kohei, Nobuko Yoshida, and Marco Carbone (2008). "Multiparty asynchronous session types". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, pp. 273–284.

Hu, Raymond, Nobuko Yoshida, and Kohei Honda (2008). "Session-Based Distributed Programming in Java". In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, pp. 516–541.

Idris (n.d.). URL: https://www.idris-lang.org/.

Idris on Github (n.d.). URL: https://github.com/idris-lang/Idris-dev.

JBoss (n.d.). URL: http://http://www.jboss.org/.

Kouzapas, Dimitrios et al. (2016). "Typechecking protocols with Mungo and StMungo". In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*. Ed. by James Cheney and Germán Vidal. ACM, pp. 146–159.

Neubauer, Matthias and Peter Thiemann (2004). "An Implementation of Session Types". In: *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*. Ed. by Bharat Jayaraman. Vol. 3057. Lecture Notes in Computer Science. Springer, pp. 56–70.

Ng, Nicholas, Nobuko Yoshida, and Kohei Honda (2012). "Multiparty Session C: Safe Parallel Programming with Message Optimisation". In: *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*. Ed. by Carlo A. Furia and Sebastian Nanz. Vol. 7304. Lecture Notes in Computer Science. Springer, pp. 202–218.

Scalas, Alceste and Nobuko Yoshida (2016). "Lightweight Session Programming in Scala". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Scribble on Github (n.d.). URL: https://github.com/scribble.

Spoofax (n.d.). URL: http://www.metaborg.org/.

Toninho, Bernardo, Luís Caires, and Frank Pfenning (2011). "Dependent session types via intuitionistic linear type theory". In: *Proceedings of the 13th International ACM SIGPLAN*

Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark. Ed. by Peter Schneider-Kamp and Michael Hanus. ACM, pp. 161–172.

Toninho, Bernardo and Nobuko Yoshida (2017). "Certifying data in multiparty session types". In: *Journal of Logic and Algebraic Programming* 90, pp. 61–83.

Tutorial, Scribble (n.d.). URL: http://www.scribble.org/docs/scribble-java.html#SCRIBCORE.

Vollebregt, Tobi, Lennart C. L. Kats, and Eelco Visser (2012). "Declarative specification of template-based textual editors". In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Ed. by Anthony Sloane and Suzana Andova. ACM, pp. 1–7.

Wu, Hanwen and Hongwei Xi (2017). "Dependent Session Types". In: *arXiv preprint arXiv:1704.07004*.

Yoshida, Nobuko, Pierre-Malo Deniélou, et al. (2010). "Parameterised Multiparty Session Types". In: *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by C.-H. Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer, pp. 128–145.

Yoshida, Nobuko, Raymond Hu, et al. (2013). "The Scribble Protocol Language". In: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*. Ed. by Martín Abadi and Alberto Lluch-Lafuente. Vol. 8358. Lecture Notes in Computer Science. Springer, pp. 22–41.

®

Appendix A

Idris Generation Defaults

The Idris code generation rules as explained in chapter 4 all make use of the same names and constructions to create the required code. For brevity, these defaults have been left out in the examples. For completion, this section will expand on the missing elements.

A.1 Interface

Figure A.1 shows the full rule for generating the function signature for a sending interaction. The patterns of the rule are similar to almost all other interaction translation rules. This will be discussed in appendix A.1.1.

Except for line 8, this rule uses only default values and constructions in the `with` clause, all shared by one or more of the other translation rules. Each of these clauses is explained in appendix A.1.2.

A.1.1 Rule patterns

The left- and right-hand side patterns of a Stratego rule represent the transformation from one pattern to another. For the code generation rules, a number of arguments are joined in a tuple to provide the information to generate the correct code.

```
1  idr-if: ([LISend(msg,receivers)|ints], [state|states], reqVar) ->
2              [newFunction | <idr-if> (ints,states,reqVar)]
3    with
4      stateType     := <lookup> ("statetype", reqVar)
5    ; stateHolder   := <lookup> ("stateholder", reqVar)
6    ; mName         := <lookup> ("mName", reqVar)
7    ; defaultParams := [FParamDecl(VarDecl(stateHolder, "Var"))]
8    ; payloadParams := <create-params> (msg,state)
9    ; <?(curState, curParams)> state
10   ; preState  := PreState(stateType, ParametDElement(curState,
11       <state-param-names> curParams))
12   ; (nextState,nextParams) := <next-state> states
13   ; postState := PostState(stateType,ParametDElement(nextState,
14       <state-param-names> nextParams))
15   ; newStateDecl := FStateChange(stateHolder, preState, postState)
16   ; newState     := FState(mName, IEmptyReturn(), newStateDecl)
17   ; finalParams  := <concat> [defaultParams, payloadParams]
18   ; newFunction  := ParamFunctionDecl(<msg-name> msg, finalParams,  newState)
```

Figure A.1: Stratego rule for sending interaction

The left-hand side consists of a list with at the head the interaction type and its arguments, in the case of figure A.1 a send interaction (`LISend(msg,receivers)`). The tail of the list is the remainder of the interactions. The second argument is a list of the states, with the pre-state of the current interaction at the head. The simultaneous traversal of these lists, in combination with their symmetry, ensures that the correct states are used as pre- and post-states for the interaction. The final argument is a list of required variables for the proper construction of function signatures and calls, `reqVar`. This list will come up when a variable shared by the whole program is required, a number of which will be seen in the next section.

The right-hand side of the rule represents the resulting pattern. For the interface, this is a list of interactions, with the result of the current interaction, `newFunction`, at the head, followed by the results of applying the interface generation rule `idr-if` to the remaining interactions. The tails of both the list of interactions and states are used as arguments, as well as the list of required variables.

A.1.2 `with` terms

The goal for a rule like the interface generation rule `idr-if` is to take a Scribble interaction and transform it into an Idris construct. This transformation requires the construction of a list of function parameters, possibly a return type, and of course the pre- and post-states. This information is constructed in the `with` clause of the generation rules. The rule in figure A.1 comes as close to a 'default' construction rule as possible: it uses a default parameter, has an empty return type, and it simply transitions from one state to the next. As such, it is a useful rule to use to expand on the common clauses used in these types of rules, which are omitted in the examples in the main report.

First, some basic variables are gathered from the universally used `reqVar` to ensure the code remains consistent on lines 4 to 6. The next line generates the default parameter that all stateful functions need: the stateholder. This is a function parameter with the name of the stateholder and its type, `Var`. Line 8 is skipped in this section, since it is unique to the send interaction rule.

Next is one of the most important elements of a function signature: its pre-state. First, the current state name and associated parameters are gathered from the `state` argument in the rule's left-hand side. Section 4.1 showed that, after naming the states, a state is represented by a tuple of its name and parameters. Line 9 extracts these by pattern matching this tuple. The results are then used to construct a ScIdris `PreState` construct representing an `ST` pre-state. It uses the type of the state (e.g. `StateT`) and a list consisting of the names of the associated parameters as arguments, resulting in, for example, `StateT (C4 p)`. The post-state is constructed similarly, but the state name and parameters are extracted using the `next-state` rule. This rule is required to handle the different forms the `states` tail may have. For example, the next state might be nested in a parent list, or it may be empty if this is the last interaction.

The constructed pre- and post-states can be used to define a state transition, in the default case a transition from one state to one other, represented by `FStateChange`. Line 15 shows that this uses the name of the stateholder along with the states to create this transition. An example of a result may be `[l ::: StateT (C3) :-> StateT (C4 p)]`. This transition is then used to construct the whole `ST` construct on line 16, along with the computation context name and the empty return construct.

Line 17 joins the default parameters with the specially constructed payload-dependent parameters into one list of parameters used to create the final parameterized function declaration. `ParamFunctionDecl` takes a function name, a list of parameters, and an `ST` state declaration as arguments to construct a complete function signature.

The constructs explained here are often used in the other code generation rules. As such, when some lines are missing from a `with` or `where` clause for brevity, one may assume that the rules listed here are used.

Appendix B

Interface Generation Rules

B.1 Sending Signature Generation

```
1  idr-if: ([LISend(msg,receivers)|ints], [state|states], reqVar) ->
2              [newFunction | <idr-if> (ints,states,reqVar)]
3    with
4      /* ... */
5    ; payloadParams := <create-params> (msg,state)
6    ; finalParams  := <concat> [defaultParams, payloadParams]
7    ; newFunction  := ParamFunctionDecl(<msg-name> msg, finalParams,  newState)
```

(a) Stratego rule for sending interactions

```
1  create-params: (Message(name,Some(Load(load))),state) -> <create-params> (load,state)
2  create-params: (Message(name,None()),state) -> []
3  create-params: ([PLType(vartype) | load],state) ->
4    [FParam(VVarName(vartype)) | <create-params> (load,state)]
5  create-params: ([PLDeclaration(VariableDecl(varname,vartype))|load],state) ->
6    [FParamDecl(VarDecl(varname,vartype)) | <create-params>(load,state)]
7  create-params: ([PLName(varname) | load],state@(statename,stateVars)) ->
8    [FParamDecl(VarDecl(varname,varType)) | <create-params> (load,state)]
9    with varType := <lookup> (varname,stateVars)
10 create-params: ([],_) -> []
```

(b) `create-params` rule

Figure B.1: Rules used for generating sending interaction signatures

There is just a single rule for a sending interaction, since only the function parameters are affected, rather than the whole structure. Figure B.1(a) shows this rule, with the default parameters discussed in appendix A now omitted. Creating the list of parameters from the message payload is done using the `create-params` rule, called on line 5.

The `create-params` rule takes a message with either no load (`None()`) or some payload (`Some(Load(load))`). In case of the former, no extra parameters are required and an empty list is returned, as seen on line 2. In case of the latter, a distinction is made between the three remaining payload types. Line 3 shows that if only a generic type is specified (`TypeDecl(type)`), said type is used as a parameter. This results in an unnamed parameter for the function signature. Line 5 pertains to new invariant declarations. These contain the name and type (`varname` and `vartype`, respectively) of a new invariant, which are combined and used as a named parameter (e.g. `(p : Integer)`). For known invariants, whose payload only consists of a variable name, the type must first be found. Since the invariant is

known, its type information is contained in the state information. A simple `lookup` should therefore return the type, and a named parameter can be added to the list of parameters.

B.2 Receiving Signature Generation

```
idr-if: ([LIRecv(msg,sender)|ints], [state|states],reqVar) ->
    [newFunction | <idr-if> (ints,states,reqVar)]
  where <?Message(name,None())> msg
  with
    /* ... */
  ; newFunction := ParamFunctionDecl(<msg-name> msg, finalParams, newState)
```

(a) Receiving interaction with no payload

```
idr-if: ([rcv@LIRecv(msg,sender)|ints], s@[state|states],reqVar) ->
    [newFunction | <idr-if> (ints,states,reqVar)]
  where <?Message(name,Some(payload))>msg
  with
    newPayloads := <collect(?VariableDecl(_,_))> payload
  ; knownPayloads := <collect-all(?PLName(_))> payload
  ; anonPayloads := <collect-all(?PLType(_))> payload
  ; newFunction := <idr-if-recv> (rcv, s, reqVar, newPayloads, knownPayloads,
    anonPayloads)
```

(b) Receiving interaction with payload

Figure B.2: Stratego rules for receiving interactions

The function description for a receiving interaction is dependent on the payload it carries. In case no payload is expected, a default message interaction signature is created, as shown in Figure B.2(a). Otherwise, the different payload types (new invariants, known references, and anonymous types) are gathered and used as arguments to call the specific `idr-if-recv` rule. This rule has three cases, shown in figures B.3 to B.5. The first case handles incoming messages that have one or more new invariants, and an arbitrary number of other payload types. A special exception is made for when a message has only one new invariant, and no other payload. This case is shown in figure B.4. The other case, shown in figure B.5, handles the case where no new invariants are declared, but only other payload types are present.

When multiple payload types are present in an incoming message, it needs a return type that can distinguish them. Each payload type affects the return type and the post-state. The function's return type is a tuple containing the information of the payload types: simple types for new invariants and anonymous types, and dependent pairs for known invariant references. The post-state is dependent on the payload as well, as the pattern of `res` is used to capture new invariants, as described in 2.2.2. The rule in figure B.3 captures this logic. It first collects the names and types of all payload types on lines 7 to 11. It creates the dependent pairs for the known invariants using `get-dependent-ret-pairs`, shown in figure B.6. It also creates two lists of wildcards (_) from the known and anonymous types on lines 13 and 14, to be used in the post-state. The tuple that is the final return type is created on line 15. The list of names and wildcards from line 16 is used to pattern match the result and extract the invariant names, so that they can be used as arguments for the post-state.

When only one new invariant is received, and no other payload, the return type is a simple type and a smaller post-state is created.

When no new invariants are introduced at all, the post-state is the default style, and only a tuple of return types need to be created, as is shown in figure B.5.

```
idr-if-recv: (LIRecv(msg,sender), [state|states],reqVar, newPayloads, knownPayloads,
    anonPayloads, ints) -> newFunction
  where
    not(<?[_]> newPayloads;<?[]> knownPayloads;<?[]> anonPayloads)
  ; not(<?[]> newPayloads)
  with
    /* ... */
    newReturnNames := <get-payload-names> newPayloads
  ; newReturnTypes := <get-payload-types> newPayloads
  ; knwReturnNames := <get-payload-names> knownPayloads
  ; knwReturnTypes := <get-payload-types> (knwReturnNames,
  ; anonReturnTypes := <get-payload-types> anonPayloads curParams)
  ; knwReturnDPairs:= <get-dependent-ret-pairs> (knwReturnNames, curParams)
  ; knwReturnNamesWildCards := <to-wildcards> knwReturnNames
  ; anonReturnWildCards := <to-wildcards> anonReturnTypes
  ; returnTypes := IReturn(VarType(Tuple(<concat> [newReturnTypes, knwReturnDPairs,
    anonReturnTypes])))
  ; returnNames := VarType(Tuple(<concat> [newReturnNames, knwReturnNamesWildCards,
    anonReturnWildCards]))
  ; stateCaseDecl := CasePure(returnNames, ParametDElement(nextState, <state-param-names>
    nextParams))
  /* ... */
  ; newFunction := ParamFunctionDecl(functionName,defaultParams,newState)
```

Figure B.3: Stratego rule for receiving interaction with multiple payload types

```
idr-if-recv: (LIRecv(msg,sender), [state|states],reqVar, newPayloads, knownPayloads,
    anonPayloads, ints) -> newFunction
  where
    <?[VariableDecl(var,type)]> newPayloads
  ; <?[]> knownPayloads
  ; <?[]> anonPayloads
  with
    /* ... */
    returnType := IReturn(SimpleType(type))
  . /* ... */
  ; newFunction := ParamFunctionDecl(functionName,defaultParams,newState)
```

Figure B.4: Stratego rule for receiving interaction with exactly one new invariant

```
idr-if-recv: (LIRecv(msg,sender), [state|states],reqVar, newPayloads, knownPayloads,
    anonPayloads, ints) -> newFunction
  where
    <?[]> newPayloads
  with
    /* ... */
    knwReturnNames := <get-payload-names> knownPayloads
  ; knwReturnTypes := <get-payload-types> (knwReturnNames, curParams)
  ; knwReturnDPairs:= <get-dependent-ret-pairs> (knwReturnNames, curParams)
  ; knwReturnNamesWildCards := <to-wildcards> knwReturnNames
  ; returnTypes := IReturn(VarType(Tuple(knwReturnDPairs)))
  /* ... */
  ; newFunction := ParamFunctionDecl(functionName,defaultParams,newState)
```

Figure B.5: Stratego rule for receiving interaction with no *new* payload

```
get-dependent-ret-pairs: ([VVarName(name)|names],params) ->
    [depPair | <get-dependent-ret-pairs> (names,params)]
  with
    type := <lookup> (name,params)
  ; newName := <concat-strings> [name,"'"]
  ; depPair := DepPair(newName,type,newName,name)
get-dependent-ret-pairs: ([],_) -> []
```

Figure B.6: Stratego rule for creating dependent pairs

B.3 Choice Signature Generation

```
1  idr-if: ([LNIChoice(name,role,blocks)|ints], [state,blockStates|states], reqVar) ->
2    <concat> [[newFunction], blocksRes, intsRes]
3  with
4  ; [postCase|postCases] := <gen-choice-posts> (blockStates,choices)
5  ; postState := CPostState(stateType, postCase,postCases)
6  /* ... */
7  ; newFunction := ParamFunctionDecl(name,defaultParams,newState)
8  ; blocksRes := <idr-if> (blocks,blockStates,reqVar)
9  ; intsRes := <idr-if> (ints,states,reqVar)
10
11 gen-choice-posts: ([block|blocks], [choice|choices]) ->
12   [<gen-choice-case> (block,choice) | <gen-choice-posts> (blocks,choices)]
13 where not(<?(_,_)> block)
14 gen-choice-posts: ([],[_|_]) -> CasePure(WildCard(), ParametDElement("Done",[]))
15 gen-choice-posts: ([],[]) -> []
16
17 gen-choice-case: ([(curState,curParams)|_], DataElement(choice)) ->
18   CasePure(choice, ParametDElement(curState, <state-param-names> curParams))
```

Figure B.7: Stratego rules for choice interaction

Figure B.7 shows the Stratego rule for generating the Idris code for a choice interaction, using the logic explained in section 4.3.2. The states passed as an argument on line 1, `[state,blockStates|states]`, represent the structure of the environment a choice interaction creates: the `state` is the pre-state of the choice; `blockStates` is the list containing the environments for each choice option; and `states` represents the remainder of the states for the remaining interactions. Line 4 calls the `gen-choice-posts` to gather the states in which the function might end up, using the `blockStates` as an argument.

`gen-choice-posts` extracts the first state of each list in `blockStates` and associates it with a choice result, represented by `choice`[1], which it then wraps up in a construct that can be used to generate the full, return-type dependent post-state. It does so by traversing the blocks of states of which `blockStates` consists and calling `gen-choice-case` for each. This rule returns a case construct, as seen in section 4.2, which has an argument for the pattern matching of the `case` statement (e.g. `Option1`), and a resulting state (e.g. `C3 p`).

The second case for `gen-choice-posts`, where `([],[_|_])` is expected, handles a case which may occur due to an imperfection in the generation of the `Choice` datatype: Only a single one of these types is generated for one protocol, which consists of a number of `Options` equal to the maximum number of options of any one choice interaction within the protocol. As such, it may occur that a `case` statement for a choice interaction which contains only two options, pattern matches using a `Choice` data type which contains three or more options. As such, a wildcard is used to match the remaining cases and transition to the final `Done` state, effectively ending the protocol. The Idris type checker requires this complete matching, but the return type should always match one of the named cases.

After all the cases have been generated, they are used to create the post-state on line 5. The new function is generated in the default steps, ending in its declaration on line 7. After this, the choice's blocks and the interactions following them can be processed as well.

[1]Here, `choice` represents an element of the `Choice` data type explained in chapter 2.

Appendix C

Protocol Generation

C.1 Protocol Generation Initiation

```
idr-protocol: (p@LocalProtDec(hdr,block), E, reqVar'') -> translBlock
with
  // Update for the new environment applicable to this protocol
  reqVar' :=  <update> ("environment", E, reqVar'')
; stateHolder := <lookup> ("stateholder", reqVar')
; <?(LProtHeader(protName,_))> hdr
; uniqueProtName := protName
; defaultFunctions := <default-p-functions> (uniqueProtName, E, reqVar')
; protBlock := AFunctionImpl(FunctionImpl(uniqueProtName, [VVarName(stateHolder)],
    FunctionBlock([LineDo(FunctionBlock(<idr-pr> (block,E,reqVar')))])))
; reqVar := <update> ("doDone",2,reqVar')
; recBlocks := <collect-recursion-blocks> (block, E, reqVar)
; translBlock := <declarations-first> <concat> [defaultFunctions, [protBlock],
  recBlocks]
```

C.2 Sending Calls

```
1  idr-pr: ([LISend(msg,receivers)|ints], [state|states], reqVar) ->
2    <concat> [invDecls,[newCall], <idr-pr> (ints,states,reqVar)]
3  where
4    newInvs := <collect-all(?VariableDecl(_,_))> msg
5  ; not(<?[]> newInvs)
6  with
7    /* ... */
8    invDecls := <declare-new-invariants> newInvs
9  ; localParams := <to-vvar> <to-send-params> msg
10 ; finalParams := <concat> [defaultParams, localParams]
11 ; newCall := LFunction(FunctionCall(functionName,finalParams))
12
13 declare-new-invariants: [VariableDecl(name,type) | decls] ->
14   [Let(VVarName(name),<default-type-value> type) | <declare-new-invariants> decls]
15 declare-new-invariants: [] -> []
```

Figure C.1: Stratego rules for sending interaction

A sending interaction is generally a simple call: it consists of the name of the message and the names of the required arguments. These can be found easily in the message interaction and adjoined states. There is, however, one exception to this rule. When a new invariant is declared by a sending interaction, this invariant's value must be set by the sender. Figure C.1 shows the rule for a sending interaction with at least one new invariant, collected on line 4. On line 8, rule `declare-new-invariants` is invoked with the list of invariants as an argument. This rule, listed on line 13, creates a `Let` structure for each new invariant, using its name and a hard-coded default as a value. This results in, for example, an Idris line stating `let p = 10`. Such a line is created for each new invariant, so that all of them have a value. Line 9 converts the parameters named in the message to a list of names used as arguments for the function. These, combined with the default parameters (in this case `"l"` for the state-holder), serve as an argument in the creation of the function itself on line 11. The result of the rule, shown on line 2, is a list of the new `let` bindings, the call to the send interaction, and the result of processing the remaining interactions.

C.3 Receiving Calls

```
1  idr-pr: ([LIRecv(msg,sender)|ints], [state|states],reqVar) -> [newCall | <idr-pr>
     (ints,states,reqVar)]
2  where
3    <?Message(name,None())> msg \label{line:apdxGenProtRecNoneNone}
4  with
5    stateholder := <lookup> ("stateholder", reqVar)
6  ; defaultParams := [VVarName(stateholder)]
7  ; newCall := LFunction(FunctionCall(name,defaultParams))
```

(a) Payload is empty

```
1  idr-pr: ([rcv@LIRecv(msg,sender)|ints], s@[state|states],reqVar) ->
2    [newCall | <idr-pr> (ints,states,reqVar)]
3  where
4    <?Message(name,Some(payload))> msg
5  ; newPayloads := <collect(?VariableDecl(_,_))> payload
6  ; knownPayloads := <collect-all(?PLName(_))> payload
7  ; anonPayloads := <collect-all(?PLType(_))> payload
8  with
9    newCall := <idr-pr-recv> (rcv, s, reqVar, newPayloads, knownPayloads, anonPayloads)
```

(b) Payload is not empty

Figure C.2: Stratego rule for receiving interactions

The result of receiving calls depends mostly on its payload. Figure C.2(a) shows the rule for an empty incoming message, indicated on line 3 by payload `None()`. In this case, the return is simply a function call with the stateholder as a parameter.

In case the message does contain a payload, the `idr-pr-recv` rule is called with all three types of payload (new declarations, known references, and anonymous types) as added arguments. Figure C.3 shows the three cases for this rule.

These separate cases exist only for creating visually pleasant code. A general rule could be made that would generate code that would work for all cases, but this would decrease readability of the Idris code. The first case contains two or more new invariants, and possibly other payload. The second case has no *new* invariant information, but perhaps known or anonymous payload. The third case covers the possibility when a message contains *only* one new invariant, and no other information.

The first case is shown in Figure C.3(a), where the message contains multiple payloads. The `where` clauses specify that it must not have an empty `newPayloads`, and it must not have *only* one new invariant declaration with no other payloads. In other words, it must have two or more new invariant declarations, and an arbitrary amount of known and anonymous payloads. This rule generates a call where the return type is a tuple of the new invariants, followed by a number of wildcards (_ in Idris) equal to the sum of known and anonymous invariants. This is because the known invariants are already familiar to the actor and do not need to redeclare the name, and the anonymous types remain anonymous in this code generation.

The second case, shown in Figure C.3(b), covers the case where no new invariant is declared. In this case, since the other two payload types do not need a return type, no return type at all is expected, and this is only a simple call to the function.

The third case is when the message contains only one new invariant, and no other information. In this case, the return value is one new invariant.

```
idr-pr-recv: (LIRecv(msg,sender), [state|states],reqVar, newPayloads, knownPayloads,
  anonPayloads) -> newCall
where
  not(<?[]> newPayloads)
; not(<?[VariableDecl(_,_)]> newPayloads ; <?[]> knownPayloads ; <?[]> anonPayloads)
with
  /* ... */
; newReturnNames := <get-payload-names> newPayloads
; wildcards := <to-wildcards> <concat> [knownPayloads,anonPayloads]
; returnVar := Tuple(<concat> [newReturnNames,wildcards])
; call := FunctionCall(<msg-name> msg,defaultParams)
; newCall := Assignment(AssignVar(returnVar,call))
```

(a) Payload has two or more invariants

```
idr-pr-recv: (LIRecv(msg,sender), [state|states],reqVar, newPayloads, knownPayloads,
  anonPayloads) -> newCall
where
  <?[]> newPayloads
with
  stateHolder := <lookup> ("stateholder", reqVar)
; defaultParams := [VVarName(stateHolder)]
; newCall := LFunction(FunctionCall(<msg-name> msg, defaultParams))
```

(b) Payload has no *new* invariants

```
idr-pr-recv: (LIRecv(msg,sender), [state|states],reqVar, newPayloads, knownPayloads,
  anonPayloads) -> newCall
where
  <?[VariableDecl(var,type)]> newPayloads
; <?[]> knownPayloads
; <?[]> anonPayloads
with
/* ... */
; returnVar := VVarName(var)
; newCall := Assignment(AssignVar(returnVar,FunctionCall(<msg-name> msg,defaultParams)))
```

(c) Payload has only exactly one new invariant

Figure C.3: Stratego rule for receiving interactions with invariant payload

C.4 Choice Interactions

```
idr-pr: ([LNIChoice(name,role,blocks)|ints], [state,blockStates|states], reqVar)
-> [newCall,case|<idr-pr> (ints,states,reqVar')]
with
  /* ... */
  //Create initial function call
  returnVar := VVarName("choiceResult")
; newCall := Assignment(AssignVar(returnVar,FunctionCall(name,defaultParams)))
  // Create cases
; Data(_,choices) := <lookup> ("choice-data", reqVar)
; cases := <create-choice-cases> (blocks, blockStates, choices, reqVar)
; case := JustNothing(Case(returnVar,cases))
; reqVar' := <update> ("doDone",0,reqVar)
```

(a) Choice interaction

```
create-choice-cases: ([block@LPBlock(_) | blocks], [blockStates|states],
  [DataElement(choice)|choices], reqVar)
  -> res
with
  blockRes := LineDo(FunctionBlock(<idr-pr> (block, blockStates, reqVar)))
; curChoice := LinePure(VVarName(choice), blockRes)
; res := [curChoice | <create-choice-cases> (blocks, states, choices, reqVar)]
create-choice-cases: ([],[],[choice|choices],_) ->
  [LinePure(WildCard(), LineDo(LFunction(PutStr(VVarName("Incorrect option picked")))))]
create-choice-cases: ([],[],[],_) -> []
```

(b) Case generation

Figure C.4: Stratego rules for choice interactions

C.5 Recursion

```
idr-pr: ([LIRecursion(recVarName, block)|ints], [blockStates|states], reqVar)
-> [newCall | <idr-pr> (ints,states,reqVar)]
with
  stateHolder := <lookup> ("stateholder", reqVar)
; newCall := LFunction(FunctionCall(<concat-strings>
  ["recursion_",recVarName],[VVarName(stateHolder)]))
```

Figure C.5: Stratego rule for recursion interaction call

```
collect-recursion-blocks: ([rec@LIRecursion(recVarName, block)|ints],
  [blockStates|states], reqVar)
-> <concat> [newFuncs, <collect-recursion-blocks> (ints,states, reqVar)]
with
  /* ... */
  <?[(curState,curParams)|_]> blockStates
; (nextState,nextParams) := <next-state> states
  /* ... */
  // Create function signature
; newFunctionSig :=
  AFunctionDecl(ParamFunctionDecl(functionName,defaultParams,newState))
  // Create function implementation
; newFunctionImpl := AFunctionImpl(FunctionImpl(functionName, [VVarName(stateHolder)],
  FunctionBlock([LineDo(FunctionBlock(<idr-pr> (block,blockStates,reqVar)))])))
; newFuncs := [newFunctionSig,newFunctionImpl]
```

Figure C.6: Stratego rule for recursion implementation

Appendix D

Test Cases

D.1 Choice Tests

```
module choices;

/* Multiple choices, with different number of options */
global protocol MoreChoices(role A, role B){
  choice at A {
    a1() from A to B;
  } or {
    a2() from A to B;
  } or {
    a3() from A to B;
    choice at B {
      a4() from B to A;
    } or {
      a5() from B to A;
    }
  }
  choice at B {
    a6() from B to A;
  } or {
    a7() from B to A;
  }
}

global protocol InteractionsAfterChoice(role A, role B){
  choice at A {
    b1() from A to B;
  } or {
    b2() from A to B;
  }
  b3() from B to A;
}

global protocol ActorNotParticipating(role A, role B, role C){
  c1() from A to B;
  c2() from A to C;
  choice at A {
    c3() from A to B;
  } or {
    c4() from A to B;
  }
  c5 from A to C;
```

```
}
```

D.2 Recursion Tests

```
module recursion;

global protocol Simple (role A, role B){
  a1() from A to B;
  rec ra {
    a2() from B to A;
    a3() from A to B;
  }
}

global protocol FirstLine (role A, role B){
  rec rb {
    b1() from A to B;
  }
}

global protocol Sandwiched (role A, role B){
  c1() from A to B;
  rec rc {
    c2() from A to B;
    continue rc;
  }
  c3() from B to A;
}

global protocol NoMessage (role A, role B){h1() from A to B;rec rh continue rh;global
    protocol NoMessageChoice (role A, role B)   e1() from A to B;rec ree2() from A to
    B;choice at Acontinue re; or continue re;global protocol NestedRecursion (role A, role
    B)d1() from A to B;rec rd1 d2() from B to A;d3() from A to B;rec rd2 d4() from B to
    A;d5() from A to B;choice at A d6() from A to B;continue rd1; or d7() from A to
    B;continue rd2;global protocol MultipleRecursion (role A, role B)rec rf1f1() from A to
    B;continue rf1;rec rf2f2() from A to B;continue rf2;global protocol
    ActorNotParticipating (role A, roleB, role C)g1() from A to C;rec rgg2() from B to
    A;continue rg;g3() from C to A;
```

D.3 Payload and Invariants

D.3.1 Should Pass

```
module payload;

global protocol OneInvariant (role A, role B){
  a1() from A to B;
  a2(i : Integer) from B to A;
  a3(i) from A to B;
}

global protocol TwoInvariantsAtOnce (role A, role B){
  b1(i : Integer, s : String) from A to B;
  b2(i,s) from B to A;
}

global protocol TwoInvariants(role A, role B){
```

```
  c1(i : Integer) from A to B;
  c2(s : String) from B to A;
  c3(i,s) from A to B;
}

global protocol InvariantAndKnown(role A, role B){
  d1(i : Integer) from A to B;
  d2(i, s : String) from A to B;
  d3(i,s) from B to A;
}

global protocol TwoInvariantsMulti(role A, role B, role C){
  e1(i : Integer) from A to B;
  e2(s : String)  from B to C;
  e3(i) from A to C;
  e4(i,s) from C to A;
  dummyRequiredForEdgeCaseECUP1() from C to A;
}


global protocol InvariantIntroducedAtEnd(role A, role B){
  h1() from A to B;
  h2() from B to A;
  h3(i : Integer) from A to B;
}

global protocol InvariantIntroducedAtEnd(role A, role B, role C){
  i1(i : Integer) from A to B;
  i2(i) from B to C;
}

global protocol GenericReturn(role A, role B){
  f1(Integer) from A to B;
  f2(Integer,Integer) from B to A;
  f3(String,Integer) from A to B;
}

global protocol MultipleReturnTypes(role A, role B){
  g1(Integer,s : String, String) from A to B;
  g2(s, i : Integer) from B to A;
  g3(Integer,s,i) from A to B;
  g4(x : Integer, i) from A to B;
  g5(i,x) from A to B;
  g6(x, Integer, i) from B to A;
}
```

D.3.2 Should Fail

```
module invariantsbreak;

global protocol IllegalRef (role A, role B){
  a() from A to B;
  b(p) from B to A;
}

global protocol DoubleDec (role A, role B){
  a(i : Integer) from A to B;
```

```
  b(i : Integer) from B to A;
}

global protocol DoubleDecMulti (role A, role B, role C, role D){
  a(i : Integer) from A to B;
  b() from B to A;
  c(i : Integer) from C to D;
  d() from D to C;
}
```

D.4 Invariant Scope

D.4.1 Should Pass

```
global protocol ChoiceScope(role A, role B){
  a1(s : String) from A to B;
  choice at B {
    a2(s) from B to A;
  } or {
    a3(s) from B to A;
  }
}

global protocol ChoiceScopeTwo(role A, role B){
  choice at A{
    b1(s : String) from A to B;
    b2(s) from B to A;
  } or {
    b3(i : Integer) from A to B;
    b4(i) from B to A;
  }
}

global protocol RecursionScope(role A, role B){
  c1(s : String) from A to B;
  rec rc{
    c2(s) from B to A;
    continue rc;
  }
}

global protocol RecursionScopeTwo(role A, role B){
  rec rd{
    d1(s : String) from A to B;
    d2(s) from B to A;
    continue rd;
  }
}

global protocol BothScope(role A, role B){
  e1(s : String) from A to B;
  rec re{
    choice at B{
      e2(s) from B to A;
      e3(i : Integer) from B to A;
      e4(i) from A to B;
    } or {
```

```
      e5(s) from B to A;
      continue re;
    }
  }
  e6(s) from A to B;
}
```

D.4.2 Should Fail

```
module scoping;

global protocol ChoiceScope(role A, role B){
  a1(i : Integer) from A to B;
  choice at A{
    a2(s : String) from A to B;
    a3(i,s) from B to A;
  } or {
    a4() from A to B;
    a5(i,s) from B to A;
  }
}

global protocol ChoiceScopeTwo(role A, role B){
  choice at A{
    b1(s : String) from A to B;
  } or {
    b2() from A to B;
  }
  b3(s) from B to A;
}

global protocol RecursionScope(role A, role B){
  rec rc {
    c1(s : String) from A to B;
  }
  c2(s) from A to B;
}
```

D.5 sec:Scenarios

D.5.1 ATM

```
module ATM;

global protocol ATM (role Machine, role Client){
  enterPIN(pin: Integer) from Client to Machine;
  rec x {
    choice at Client{
      pressDeposit() from Client to Machine;
      insertMoney(amt : Integer) from Client to Machine;
      updatedBalance(bal : Integer) from Machine to Client;
      cont x;
    } or {
      pressWithdraw() from Client to Machine;
      chooseAmount(amt : Integer) from Client to Machine;
```

```
      provideMoney() from Machine to Client;
      updatedBalance1(bal : Integer) from Machine to Client;
      cont x;
    } or {
      pressBalance() from Client to Machine;
      showBalance(bal : Integer) from Machine to Client;
      cont x;
    } or {
      quit() from Client to Machine;
    }
  }
  quitProt() from Machine to Client;
}
```

D.5.2 SEPA Credit Transfer Protocol

```
module SEPA_CTP;

global protocol CreditTransfer(role Originator, role OBank, role CnS) {
  sendInstruction(amount : Integer) from Originator to OBank;
  choice at OBank{
    //Reject
    reject() from OBank to Originator;
    dummy() from OBank to CnS;
  } or {
    //Accept
    debit() from OBank to Originator;
    sendCredit() from OBank to CnS;
    choice at CnS {
      //Reject
      rej() from CnS to OBank;
      refundCredit() from OBank to Originator;
    } or {
      //Accept
      acc() from CnS to OBank;
      choice at CnS {
        //Late return
        returnRequest() from CnS to OBank;
        refundCredit() from OBank to Originator;
      }
      or {
        finalize() from CnS to OBank;
        dummy() from OBank to Originator;
      }
    }
  }
}
```

D.5.3 SEPA Direct Debit Protocol

```
module SEPA_DDB;

global protocol DirectDebit(role Creditor, role CBank, role CnS){
  query(collAmount : Integer) from Creditor to CBank;
  choice at CBank {
    //Reject the collection
```

```
7       rejectCCollection() from CBank to Creditor;
8       dummy() from CBank to CnS;
9     } or {
10      //Continue
11      query() from CBank to CnS;
12      choice at CnS {
13        //Reject the collection
14        rej() from CnS to CBank;
15        rej() from CBank to Creditor;
16      } or {
17        //Continue
18        acc() from CnS to CBank;
19        credit() from CBank to Creditor;
20        choice at CnS {
21          //Receive a reject anyway
22          lateReject() from CnS to CBank;
23          debitRefund() from CBank to Creditor;
24        } or {
25          //Receive a refund request (should be within certain time)
26          refund() from CnS to CBank;
27          debit() from CBank to Creditor;
28        } or {
29          //Time for options expired
30          finalize() from CnS to CBank;
31          dummy() from CBank to Creditor;
32        }
33      }
34    }
35  }
```

Appendix E

Running Example

E.1 Scribble Protocol

```
module Shopping;

global protocol Shopping(role C, role S, role B)
{
  rec r {
    reqPrice() from C to S;
    priceInfo(p : Integer) from S to C;
    choice at C
    {
      dummy() from C to B;
      dummy2() from B to S;
      continue r;
    } or {
      reqTransfer(p) from C to B;
      ackTransfer(p) from B to S;
    }
  }
}
```

E.2 Idris Implementation (handmade)

```
import Control.ST
import Control.ST.ImplicitCall
import Control.IOExcept
import Network.Socket
import Network
import System
import Threads
import Data.String

{- Starting order:
    - Seller
    - Customer
    - Bank
   Command:
:exec (unsafePerformIO {ffi=(MkFFI C_Types String String)} (runIOExcept {err=String} (run
    {m=IOExcept String} Seller.protocol)))
-}

```

```
18 -- Some convenience boilerplating
19
20 interface ConsoleExcept (e : Type) (m : Type -> Type) where
21   raise : e -> m a
22
23 implementation ConsoleExcept e (IOExcept e) where
24   raise e = ioe_fail e
25
26 data Choice = Option1 | Option2
27
28 --------------
29 -- CUSTOMER --
30 --------------
31
32 namespace Customer {
33
34   data State = Ready | C1 | C2 Integer | C3 Integer | C4 Integer | C5 Integer | Done
35
36   interface CSB_Customer (m : Type -> Type) where
37     StateT      : Customer.State -> Type
38     start       : ST m Var [add (StateT Ready)]
39
40     reqPrice    : (l : Var) -> String -> ST m () [l ::: StateT Ready :-> StateT C1]
41
42     priceInfo   : (l : Var) -> ST m Integer [l ::: StateT C1 :-> \p => StateT (C2 p)]
43
44     choice      : (l : Var) -> (p : Integer) -> ST m Choice [l ::: StateT (C2 p):->
45                       \res => StateT (case res of
46                         Option1 => (C3 p)
47                         Option2 => (C5 p))]
48
49     dummy       : (l : Var) -> ST m () [l ::: StateT (C3 p) :-> StateT (C4 p)]
50
51     continue_r  : (l : Var) -> ST m () [l ::: StateT (C4 p) :-> StateT Ready]
52
53     reqTransfer : (l : Var) -> (p : Integer) -> ST m () [l ::: StateT (C5 p) :-> StateT
       Done]
54
55     done        : (l : Var) -> ST m () [remove l (StateT Done)]
56
57     failure     : String -> STrans m t xs r
58
59   using (ConsoleExcept e, ConsoleIO io, CSB_Customer io)
60     protocol : ST io () []
61
62     recursion_r : (l : Var) -> ST io () [l ::: StateT {m=io} Ready :-> StateT {m=io} Done]
63
64     protocol =
65       do cust <- start
66          recursion_r cust
67          done cust
68
69     recursion_r cust =
70       do putStr "Enter product name: \n"
71          name <- getStr
72          reqPrice cust name
73          p <- Customer.priceInfo cust
74          cr <- choice cust p
```

```
75          case cr of
76            Option1 =>
77                    do dummy cust
78                       continue_r cust
79                       recursion_r cust
80            Option2 =>
81                    do reqTransfer cust p
82                       pure()
83
84  implementation (Sockets io, ConsoleIO io, ConsoleExcept String io, Monad io) =>
      CSB_Customer io where
85    StateT Ready    = Composite[Sock {m=io} Open,Sock {m=io} Open]
86    StateT C1       = Composite[Sock {m=io} Open,Sock {m=io} Open]
87    StateT (C2 _)   = Composite[Sock {m=io} Open,Sock {m=io} Open]
88    StateT (C3 _)   = Composite[Sock {m=io} Open,Sock {m=io} Open]
89    StateT (C4 _)   = Composite[Sock {m=io} Open,Sock {m=io} Open]
90    StateT (C5 _)   = Composite[Sock {m=io} Open,Sock {m=io} Open]
91    StateT Done     = Composite[Sock {m=io} Closed,Sock {m=io} Closed]
92
93    start =
94      do Right sockS <- socket Stream | Left _ => Customer.failure "could not open socket"
95         Right _ <- connect sockS (Hostname "localhost") 9442 | Left _ =>
      Customer.failure "could not connect socket"
96         Right sockB <- socket Stream | Left _ => Customer.failure "could not open socket
      B"
97         Right _ <- bind sockB Nothing 9443 | Left _ => Customer.failure "could not bind
      socket B"
98         Right _ <- listen sockB | Left _ => Customer.failure "could not listen socket B"
99         Right sockB' <- accept sockB | Left _ => Customer.failure "could not accept
      socket B"
100        cust <- new ()
101        combine cust [sockS,sockB']
102        close sockB;remove sockB
103        putStr "Customer started\n"
104        pure cust
105
106   reqPrice cust s =
107     do [sockS,sockB] <- split cust
108        Right _ <- send sockS s | Left _ => Customer.failure "could not send"
109        combine cust [sockS,sockB]
110        pure()
111
112   priceInfo cust =
113     do [sockS,sockB] <- split cust
114        Right string <- recv sockS | Left _ => Customer.failure "could not receive"
115        putStr ("The price is: " ++ string ++ "\n")
116        case parsePositive {a=Integer} string of
117          Just x => do combine cust [sockS,sockB]; pure x
118          _      => Customer.failure "parse error"
119
120   choice cust p =
121     do [sockS,sockB] <- split cust
122        case (p > 50) of
123          True =>
124            do Right _ <- send sockS "1" | Left _ => Customer.failure "option S1 not
      sent"
125               Right _ <- send sockB "1" | Left _ => Customer.failure "option B1 not
      sent"
```

```
126                 combine cust [sockS, sockB]
127                 pure Option1
128             False =>
129               do Right _ <- send sockS "2" | Left _ => Customer.failure "option S2 not
    sent"
130                 Right _ <- send sockB "2" | Left _ => Customer.failure "option B2 not
    sent"
131                 combine cust [sockS, sockB]
132                 pure Option2
133
134     dummy cust = pure()
135
136     continue_r cust = pure()
137
138     reqTransfer cust p =
139       do [sockS,sockB] <- split cust
140          Right _ <- send sockB (cast {to=String} p) | Left _ => Customer.failure "reqT
    not sent"
141          close sockS
142          close sockB
143          combine cust [sockS,sockB]
144          pure()
145
146     done cust =
147       do [sockS,sockB] <- split cust
148          remove sockS; remove sockB
149          delete cust
150          pure ()
151
152     failure {t = t} s =
153       (>>=) {a = t} (lift (raise s)) (\_ => Customer.failure s) -- hack!
154
155 }
156
157 --------------
158 --  SELLER  --
159 --------------
160
161 namespace Seller {
162
163   data State = Ready | S1 | S2 Integer | S3 Integer | S4 Integer | S5 Integer | Done
164
165   interface CSB_Seller (m : Type -> Type) where
166
167     StateT : Seller.State -> Type
168
169     start : ST m Var [add (StateT Ready)]
170
171     reqPrice : (l : Var) -> ST m String [l ::: StateT Ready :-> StateT S1]
172
173     priceInfo : (l : Var) -> (p : Integer) -> ST m () [l ::: StateT S1 :-> StateT (S2 p)]
174
175     choice : (l : Var) -> ST m Choice [l ::: StateT (S2 p) :-> \res =>
176       StateT (case res of
177         Option1 => (S3 p)
178         Option2 => (S5 p))]
179
180     dummy2 : (l : Var) -> ST m () [l ::: StateT (S3 p) :-> StateT (S4 p)]
```

```
181
182     continue_r : (l : Var) -> ST m () [l ::: StateT (S4 p) :-> StateT Ready]
183
184     ackTransfer : (l : Var) -> ST m (p' : Integer ** p' = p)  [l ::: StateT (S5 p) :->
        StateT Done]
185
186     done        : (l : Var) -> ST m () [remove l (StateT Done)]
187
188     failure     : String -> STrans m t xs r
189
190   using (ConsoleExcept e, ConsoleIO io, CSB_Seller io)
191     protocol : ST io () []
192
193     recursion_r : (l : Var) -> ST io () [l ::: StateT {m=io} Ready :-> StateT {m=io} Done]
194
195     protocol =
196       do sel <- start
197          recursion_r sel
198          done sel
199
200     recursion_r sel =
201       do item <- reqPrice sel
202          let price = (if item == "Book" then 10 else (if item == "DVD" then 60 else 0))
203          priceInfo sel price
204          putStr "Sent priceInfo\n"
205          choiceRes <- choice sel
206          case choiceRes of
207            Option1 => do dummy2 sel
208                          continue_r sel
209                          recursion_r sel
210            Option2 => do price' <- ackTransfer sel
211                          pure()
212
213   implementation (ConsoleIO io,Sockets io, ConsoleExcept String io, Monad io) =>
       CSB_Seller io where
214     StateT Ready  = Composite [Sock {m=io} Open, Sock {m=io} Open]
215     StateT S1     = Composite [Sock {m=io} Open, Sock {m=io} Open]
216     StateT (S2 p) = Composite [Sock {m=io} Open, Sock {m=io} Open]
217     StateT (S3 p) = Composite [Sock {m=io} Open, Sock {m=io} Open]
218     StateT (S4 p) = Composite [Sock {m=io} Open, Sock {m=io} Open]
219     StateT (S5 p) = Composite [Sock {m=io} Open, Sock {m=io} Open]
220     StateT Done   = Composite [Sock {m=io} Closed, Sock {m=io} Closed]
221
222     start =
223       do Right sockC <- socket Stream | Left _ => Seller.failure "could not open socket
        C\n"
224          Right _ <- bind sockC Nothing 9442 | Left _ => Seller.failure "could not bind
        socket C\n"
225          Right _ <- listen sockC | Left _ => Seller.failure "could not listen socket C\n"
226          Right sockC' <- accept sockC | Left _ => Seller.failure "could not accept socket
        C\n"
227          Right sockB <- socket Stream | Left _ => Seller.failure "could not open socket
        B\n"
228          Right _ <- bind sockB Nothing 9444 | Left _ => Seller.failure "could not bind
        socket B\n"
229          Right _ <- listen sockB | Left _ => Seller.failure "could not listen socket B\n"
230          Right sockB' <- accept sockB | Left _ => Seller.failure "could not accept socket
        B\n"
```

```
231           close sockC; remove sockC
232           close sockB; remove sockB
233           sel <- new ()
234           combine sel [sockC',sockB']
235           putStr "Seller started\n"
236           pure sel
237
238       reqPrice sel =
239         do [sockC,sockB] <- split sel
240            Right item <- recv sockC | Left _ => Seller.failure "could not recv price\n"
241            combine sel [sockC,sockB]
242            pure item
243
244       priceInfo sel p =
245         do [sockC,sockB] <- split sel
246            Right _ <- send sockC (cast {to=String} p) | Left _ => Seller.failure "could not
      send price\n"
247            combine sel [sockC,sockB]
248            pure ()
249
250       choice sel =
251         do [sockC,sockB] <- split sel
252            Right choice <- recv sockC | Left _ => Seller.failure "could not recv choice\n"
253            combine sel [sockC,sockB]
254            case (parsePositive {a=Int} choice) of
255              Just 1 => pure Option1
256              Just 2 => pure Option2
257              _      => Seller.failure "choice not received correctly\n"
258
259       dummy2 sel = pure()
260       continue_r sel = pure()
261
262       ackTransfer {p} sel =
263         do [sockC,sockB] <- split sel
264            Right msg <- recv sockB | Left _ => Seller.failure "could not recv ack\n"
265            case (parsePositive msg) of
266              Just x =>
267                do putStr ("Received price " ++ (cast x) ++ "!\n")
268                   case decEq x p of
269                     Yes eq => do close sockC; close sockB; combine sel [sockC,sockB];
270                                  pure (x ** eq)
271                     _      => Seller.failure "invariant error"
272              _      => Seller.failure "price not received correctly\n"
273
274       done sel =
275         do [sockC,sockB] <- split sel
276            remove sockC; remove sockB;
277            delete sel
278            pure()
279
280       failure {t = t} s =
281         (>>=) {a = t} (lift (raise s)) (\_ => Seller.failure s) -- hack!
282
283   }
284
285
286   --------------
287   --   Bank   --
```

```
--------------
namespace Bank {

  data State = Ready | B1 | B2 | B3 | B4 | B5 Integer | Done

  interface CSB_Bank (m : Type -> Type) where

    StateT : Bank.State -> Type

    start : ST m Var [add (StateT Ready)]

    choice : (l : Var) -> ST m Choice [l ::: StateT Ready :-> \res =>
      StateT (case res of
        Option1 => B1
        Option2 => B4)]

    dummy : (l : Var) -> ST m () [l ::: StateT B1 :-> StateT B2]

    dummy2 : (l : Var) -> ST m () [l ::: StateT B2 :-> StateT B3]

    continue_r : (l : Var) -> ST m () [l ::: StateT B3 :-> StateT Ready]

    reqTransfer : (l : Var) -> ST m Integer [l ::: StateT B4 :-> \res => StateT (B5 res)]

    ackTransfer : (l : Var) -> (p : Integer) -> ST m () [l ::: StateT (B5 p) :-> StateT
    Done]

    done : (l : Var) -> ST m () [remove l (StateT Done)]

    failure : String -> STrans m t xs r

  using (ConsoleExcept e, ConsoleIO io, CSB_Bank io)
    protocol : ST io () []

    recursion_r : (l : Var) -> ST io () [l ::: StateT {m=io} Ready :-> StateT {m=io} Done]

    protocol =
      do bk <- start
         recursion_r bk
         done bk

    recursion_r bk =
      do cr <- choice bk
         case cr of
           Option1 =>
             do dummy bk
                dummy2 bk
                continue_r bk
                recursion_r bk
           Option2 =>
             do p <- reqTransfer bk
                ackTransfer bk p

  implementation (ConsoleIO io, Sockets io, ConsoleExcept String io, Monad io) =>
    CSB_Bank io where
    StateT Ready = Composite [Sock {m=io} Open, Sock {m=io} Open]
    StateT B1    = Composite [Sock {m=io} Open, Sock {m=io} Open]
    StateT B2    = Composite [Sock {m=io} Open, Sock {m=io} Open]
```

```
344     StateT B3    = Composite [Sock {m=io} Open, Sock {m=io} Open]
345     StateT B4    = Composite [Sock {m=io} Open, Sock {m=io} Open]
346     StateT (B5 p)= Composite [Sock {m=io} Open, Sock {m=io} Open]
347     StateT Done  = Composite [Sock {m=io} Closed, Sock {m=io} Closed]
348
349     start =
350       do Right sockC <- socket Stream | Left _ => Bank.failure "could not open socket\n"
351          Right _ <- connect sockC (Hostname "localhost") 9443 | Left _ => Bank.failure
      "could not connect to customer\n"
352          Right sockS <- socket Stream | Left _ => Bank.failure "could not open socketS\n"
353          Right _ <- connect sockS (Hostname "localhost") 9444 | Left _ => Bank.failure
      "could not connect to seller\n"
354          bk <- new ()
355          combine bk [sockC,sockS]
356          putStr "Bank started\n"
357          pure bk
358
359     choice bk =
360       do [sockC,sockS] <- split bk
361          Right choice <- recv sockC | Left _ => Bank.failure "could not recv choice\n"
362          combine bk [sockC,sockS]
363          case (parsePositive choice) of
364            Just 1 => pure Option1
365            Just 2 => pure Option2
366            _      => Bank.failure "choice not received correctly\n"
367
368
369     dummy bk = pure()
370     dummy2 bk = pure()
371     continue_r bk = pure()
372
373     reqTransfer bk =
374       do [sockC,sockS] <- split bk
375          Right price' <- recv sockC | Left _ => Bank.failure "could not recv price\n"
376          combine bk [sockC,sockS]
377          case (parsePositive {a=Integer} price') of
378            Just price => do putStr ("Received price " ++ (cast price) ++ "!\n");pure price
379            _          => Bank.failure "could not parse price\n"
380
381     ackTransfer bk p =
382       do [sockC,sockS] <- split bk
383          Right _ <- send sockS (cast p) | Left _ => Bank.failure "could not send price\n"
384          close sockC;close sockS
385          combine bk [sockC,sockS]
386          pure()
387
388     done bk =
389       do [sockC,sockS] <- split bk
390          remove sockC; remove sockS
391          delete bk
392          pure()
393
394
395     failure {t = t} s =
396       (>>=) {a = t} (lift (raise s)) (\_ => Bank.failure s) -- hack!
397
398 }
399 -- Local Variables:
```

```
-- idris-load-packages: ("contrib")
-- End:
```

E.3 Idris Implementation (generated)

E.3.1 Seller

```
import Control.ST
import Control.ST.ImplicitCall

data Choice_choice5 = Choice_choice5_Option1 | Choice_choice5_Option2

data StateD = State1 | State2 | State3 Integer | State4 Integer | State5 Integer | Done

interface Shopping_S_IF (m : Type -> Type) where
  StateT : StateD -> Type

  startShopping_S_IF : ST m (Maybe Var) [addIfJust (StateT (State1))]

  done_Shopping_S : (x : Var) -> ST m ( ) [remove x (StateT Done)]

  reqPrice : (l : Var) -> ST m ( ) [l ::: StateT (State1) :-> StateT (State2)]

  priceInfo : (l : Var) -> (p : Integer) -> ST m ( ) [l ::: StateT (State2) :-> StateT
    (State3 p)]

  choice5 : (l : Var) -> ST m (Choice_choice5) [l ::: StateT (State3 p) :-> \res =>
    StateT
      (case res of
        Choice_choice5_Option1 =>
          (State4 p)
        Choice_choice5_Option2 =>
          (State5 p))]

  dummy2 : (l : Var) -> ST m ( ) [l ::: StateT (State4 p) :-> StateT (State1)]

  ackTransfer : (l : Var) -> ST m (((p' : Integer ** p' = p))) [l ::: StateT (State5 p)
    :-> StateT (Done)]

using (ConsoleIO io, Monad io, Shopping_S_IF io)
  startShopping_S : ST io ( ) [ ]

  Shopping_S : (l : Var) -> ST io ( ) [remove l (StateT {m=io}  State1)]

  recursion_r : (l : Var) -> ST io ( ) [l ::: StateT {m=io}  (State1) :-> StateT {m=io}
    (Done)]

  startShopping_S =
    do l <- startShopping_S_IF
       case l of
        Just s =>
          Shopping_S  s
        Nothing =>
          putStr $ "Couldn't start client"

  Shopping_S l =
    do recursion_r  l
```

```
48        done_Shopping_S  l
49
50    recursion_r l =
51      do reqPrice  l
52         p <- lift $ pure $ 0
53         priceInfo  l p
54         choiceResult <- choice5  l
55         case choiceResult of
56          Choice_choice5_Option1 =>
57            do dummy2  l
58               recursion_r  l
59          Choice_choice5_Option2 =>
60            do ackTransfer  l
61               pure()
```

E.3.2 Customer

```
1  import Control.ST
2  import Control.ST.ImplicitCall
3
4  data Choice_choice4 = Choice_choice4_Option1 | Choice_choice4_Option2
5
6  data StateD = State1 | State2 | State3 Integer | State4 Integer | State5 Integer | Done
7
8  interface Shopping_C_IF (m : Type -> Type) where
9    StateT : StateD -> Type
10
11   startShopping_C_IF : ST m (Maybe Var) [addIfJust (StateT (State1))]
12
13   done_Shopping_C : (x : Var) -> ST m ( ) [remove x (StateT Done)]
14
15   reqPrice : (l : Var) -> ST m ( ) [l ::: StateT (State1) :-> StateT (State2)]
16
17   priceInfo : (l : Var) -> ST m (Integer) [l ::: StateT (State2) :-> \res => StateT
       (State3 res)]
18
19   choice4 : (l : Var) -> ST m (Choice_choice4) [l ::: StateT (State3 p) :-> \res =>
20     StateT
21       (case res of
22         Choice_choice4_Option1 =>
23           (State4 p)
24         Choice_choice4_Option2 =>
25           (State5 p))]
26
27   dummy : (l : Var) -> ST m ( ) [l ::: StateT (State4 p) :-> StateT (State1)]
28
29   reqTransfer : (l : Var) -> (p : Integer) -> ST m ( ) [l ::: StateT (State5 p) :->
       StateT (Done)]
30
31 using (ConsoleIO io, Monad io, Shopping_C_IF io)
32   startShopping_C : ST io ( ) [ ]
33
34   Shopping_C : (l : Var) -> ST io ( ) [remove l (StateT {m=io}  State1)]
35
36   recursion_r : (l : Var) -> ST io ( ) [l ::: StateT {m=io}  (State1) :-> StateT {m=io}
       (Done)]
37
```

```
38  startShopping_C =
39    do l <- startShopping_C_IF
40       case l of
41        Just s =>
42          Shopping_C  s
43        Nothing =>
44          putStr $ "Couldn't start client"
45
46  Shopping_C l =
47    do recursion_r  l
48       done_Shopping_C  l
49
50  recursion_r l =
51    do reqPrice  l
52       p <- priceInfo  l
53       choiceResult <- choice4  l
54       case choiceResult of
55        Choice_choice4_Option1 =>
56          do dummy  l
57             recursion_r  l
58        Choice_choice4_Option2 =>
59          do reqTransfer  l p
60             pure()
```

E.3.3 Bank

```
1  import Control.ST
2  import Control.ST.ImplicitCall
3
4  data Choice_choice6 = Choice_choice6_Option1 | Choice_choice6_Option2
5
6  data StateD = State1 | State2 | State3 | State4 | State5 Integer | Done
7
8  interface Shopping_B_IF (m : Type -> Type) where
9    StateT : StateD -> Type
10
11   startShopping_B_IF : ST m (Maybe Var) [addIfJust (StateT (State1))]
12
13   done_Shopping_B : (x : Var) -> ST m ( ) [remove x (StateT Done)]
14
15   choice6 : (l : Var) -> ST m (Choice_choice6) [l ::: StateT (State1) :-> \res =>
16     StateT
17       (case res of
18         Choice_choice6_Option1 =>
19           (State2)
20         Choice_choice6_Option2 =>
21           (State4))]
22
23   dummy : (l : Var) -> ST m ( ) [l ::: StateT (State2) :-> StateT (State3)]
24
25   dummy2 : (l : Var) -> ST m ( ) [l ::: StateT (State3) :-> StateT (State1)]
26
27   reqTransfer : (l : Var) -> ST m (Integer) [l ::: StateT (State4) :-> \res => StateT
       (State5 res)]
28
29   ackTransfer : (l : Var) -> (p : Integer) -> ST m ( ) [l ::: StateT (State5 p) :->
       StateT (Done)]
```

```
30
31 using (ConsoleIO io, Monad io, Shopping_B_IF io)
32   startShopping_B : ST io ( ) [ ]
33
34   Shopping_B : (l : Var) -> ST io ( ) [remove l (StateT {m=io}  State1)]
35
36   recursion_r : (l : Var) -> ST io ( ) [l ::: StateT {m=io}  (State1) :-> StateT {m=io}
       (Done)]
37
38   startShopping_B =
39     do l <- startShopping_B_IF
40        case l of
41         Just s =>
42           Shopping_B  s
43         Nothing =>
44           putStr $ "Couldn't start client"
45
46   Shopping_B l =
47     do recursion_r  l
48        done_Shopping_B  l
49
50   recursion_r l =
51     do choiceResult <- choice6  l
52        case choiceResult of
53         Choice_choice6_Option1 =>
54           do dummy  l
55              dummy2  l
56              recursion_r  l
57         Choice_choice6_Option2 =>
58           do p <- reqTransfer  l
59              ackTransfer  l p
60              pure()
```