Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Refinement Session Types

Author:
Fangyi Zhou

Supervisor:
Prof. Nobuko Yoshida

Second Marker:
Dr. Iain Phillips

16th June 2019

Abstract

We present an end-to-end framework to statically verify multiparty concurrent and distributed protocols with refinements, where refinements are in the form of logical constraints. We combine the theory of multiparty session types and refinement types and provide a type system approach for lightweight static verification. We formalise a variant of the λ-calculus, extended with refinement types, and prove their type safety properties. Based on the formalisation, we implement a refinement type system extension for the F# language. We design a functional approach to generate APIs with refinement types from a multiparty protocol in F#. We generate handler-styled APIs, which statically guarantee the linear usage of channels. With our refinement type system extension, we can check whether the implementation is correct with respect to the refinements. We evaluate the expressiveness of our system using three case studies of refined protocols.

Acknowledgements

I would like to thank Prof. Nobuko Yoshida, Dr. Francisco Ferreira, Dr. Rumyana Neykova and Dr. Raymond Hu for their advice, support and help during the project. Without them I would not be able to complete this project.

I would like to thank Prof. Paul Kelly, Prof. Philippa Gardner and Prof. Sophia Drossopoulou, whose courses enlightened me to explore the area of programming languages. Without them I would not be motivated to discover more in this area.

I would like to thank Prof. Paul Kelly again, this time for being my personal tutor. I am very grateful to Paul, who continuously supported, encouraged and inspired me throughout the degree.

I would like to thank my family for supporting me to pursue a degree abroad. I am truly grateful for their unconditional support throughout my life.

Contents

1 Introduction **9**

1.1 Motivation . . . 9

1.2 Objectives . . . 10

1.3 Contributions and Report Structure . . . 10

2 Background **13**

2.1 Session Types . . . 13

2.1.1 Asynchronous π-calculus . . . 14

2.1.2 Binary Session Types . . . 15

2.1.3 Multiparty Session Types . . . 18

2.1.4 Scribble and Endpoint API Generation . . . 21

2.1.5 F# Session Type Provider . . . 22

2.2 Refinement Types . . . 24

2.2.1 Liquid Types . . . 24

2.2.2 F7: Refinement Types for F# . . . 25

2.2.3 F* . . . 25

2.2.4 F# Session Type Provider . . . 26

3 λ^H: A Simple Refinement Calculus **29**

3.1 Syntax . . . 29

3.2 Operational Semantics . . . 32

3.3 Erasing Refinements . . . 33

3.4 Typing . . . 34

3.4.1 Type Synthesis and Checking . . . 34

3.4.2 Well-formedness of Types and Context . . . 37

3.4.3 Subtyping and Encoding . . . 38

3.4.4 Typing under Erasure . . . 40

3.5 Type Safety . . . 42

4 FluidTypes: A Type System Implementation of λ^H **49**

4.1 Implementation of λ^H Type System . . . 49
4.1.1 Data Type Definitions . . . 50
4.1.2 Main Typing Judgements . . . 50
4.1.3 Subtyping Judgements and SMT Encodings . . . 52
4.1.4 Summary . . . 53
4.2 From F# Expression to λ^H Terms . . . 54
4.2.1 Handling Expressions and Types . . . 54
4.2.2 Summary . . . 55
4.3 Annotations for Refinement Types . . . 55
4.3.1 Annotation by Custom Attributes . . . 55
4.3.2 String-based Annotation . . . 56
4.3.3 Alternative Designs . . . 56
4.3.4 Summary . . . 57
4.4 Type System Extensions . . . 57
4.4.1 Type Aliases . . . 58
4.4.2 Records . . . 59
4.4.3 Enumerations . . . 60
4.4.4 Algebraic Data Types . . . 61

5 FLUIDSESSION: Towards Statically Verified Protocol Implementation 63
5.1 Protocol Specification with Scribble . . . 63
5.2 Obtaining the CFSM from Scribble . . . 63
5.3 API Generation . . . 64
5.3.1 A Straight-line Protocol without Refinements . . . 64
5.3.2 Adding Refinements to Payloads . . . 66
5.3.3 Adding Refinements with Non-payload Variables . . . 66
5.3.4 Adding Branches . . . 68
5.3.5 Summary . . . 72
5.4 Execution of the CFSM . . . 72
5.5 Limitations . . . 73

6 Evaluation 75
6.1 Adder Example in Section 5.3 . . . 75
6.1.1 Client . . . 76
6.1.2 Server . . . 80
6.1.3 Summary . . . 83
6.2 Two Buyers Protocol . . . 83
6.3 Example Protocols with Unsupported Refinements . . . 85

7 Conclusion **87**
7.1 Contributions . . . 87
7.1.1 Open Source Contributions to F# Libraries . . . 88
7.2 Future Work . . . 88
7.2.1 Refinements in Multiparty Session Types . . . 88
7.2.2 λ^H and FLUIDTYPES Library . . . 89

A A Basic Type System for the Simply Typed λ-calculus **95**
A.1 Types and Typing Judgements . . . 95

B Lemmas and Proofs **97**
B.1 Lemmas and Proofs for Chapter 3 . . . 97

C Syntax of Refinement Annotations **105**
C.1 Syntax of Refinement Type Annotation in F# . . . 105

D Code for Evaluation **107**
D.1 Implementation for Two Buyers . . . 107

List of Figures

1.1 Workflow for Verified Implementation of Refined Protocols 11

2.1 Processes in Asynchronous π-calculus 14
2.2 Processes in Session Calculus 16
2.3 Syntax of Session Types 17
2.4 Processes in Multiparty Session Calculus 19
2.5 Global and Session Types 20
2.6 Projection from Global Types to Session Types 21
2.7 Adder Protocol 22
2.8 Finite State Machine for role `C` for Protocol in Figure 2.7 23
2.9 Generated Java API for State 2, Role `C` (truncated) 23
2.10 Refined Adder Protocol 26

3.1 Syntax of λ^H 30
3.2 Definition of Contexts 31
3.3 Reduction Rules for λ^H 32
3.4 Typing Judgements for λ^H 35
3.5 Well-formedness Judgements for Types 37
3.6 Well-formedness Judgements of Context 37
3.7 Subtyping Judgements for λ^H 38

4.1 Function Signatures of Main Typing Judgements 50
4.2 Code Snippet of Using an Attribute 55
4.3 `abs` Function with an Annotated Refinement Type 56
4.4 Invalid Code Snippet due to a Limitation of F# Attributes 58
4.5 Type Alias Definition of Non-negative Integers 58
4.6 Valid Code Snippet Using Type Alias 58
4.7 An Example Definition of Record Type 59
4.8 An Example Definition of Record Type with Refinement 59
4.9 An Example Definition of Record Type with Data Dependency 59
4.10 Proving Sum of Two Even Numbers are Even Using Record Type 60

4.11 An Example Type Definition of Enumeration Type 61
4.12 An Example Type Definition of Discriminated Union with Refinements . 61

5.1 Adder Protocol (without Refinements) 64
5.2 Finite State Machine for Role C for Protocol in Figure 5.1 65
5.3 Generated Handler Types for role C for Figure 5.1 65
5.4 Adder Protocol (with Refinements) 66
5.5 Generated Handler Types for role C for Figure 5.4 66
5.6 Adder Protocol (with More Refinements) 67
5.7 Generated State Record Definition for State 9 67
5.8 Adder Protocol (with Branches) 68
5.9 Finite State Machine for role C for Protocol in Figure 5.8 69

6.1 Handler Type Definition for Client in Adder Protocol 76
6.2 Example Implementation of Client Role 77
6.3 Auxiliary Functions for Testing Properties 79
6.4 Alternative Implementation of Client Role 80
6.5 Handler Type Definition for Server in Adder Protocol 81
6.6 Example Implementation of Server Role 82
6.7 Two Buyer Protocol in Scribble 83
6.8 Implementation for Role B of Two Buyers Protocol 84
6.9 Accumulator Protocol in Scribble 85
6.10 Refined Accumulator Protocol in Imaginary Scribble 86

A.1 Types in λ-calculus 95
A.2 Typing Judgements for λ-calculus 95

D.1 Implementation for Role A of Two Buyers Protocol 107
D.2 Implementation for Role S of Two Buyers Protocol 108

List of Tables

2.1 Structural Congruence for Asynchronous π-calculus 15
2.2 Duality of Session Types 18

3.1 Judgements of Type System 34
3.2 Constants and their Refined Types 36

5.1 Handler Generation for a State in CFSM 72

A.1 Constants and their Basic Types 96

Chapter 1

Introduction

1.1 Motivation

With the development of computer science, distributed and concurrent programming has become increasingly prevalent in recent years. Concurrency is ubiquitous in the modern computing world, ranging from multiple cores in CPU and GPU, to server clusters in large data centres. The development has brought benefits such as speedups and scalability, but also new challenges in specifying and verifying distributed and concurrent programs.

The importance of type systems in programming languages is increasingly recognised by developers, as they provide a lightweight way for program specification and verification. Type systems provide certain guarantees about programs, when typechecking succeeds. This is often remarked upon by the slogan "Well-typed programs cannot go wrong" [32]. Although it is usually difficult to prove the absence of all software bugs, static typechecking, usually run at a compilation stage, can detect classes of programming errors, without the need to execute the actual program or write test cases. Moreover, some type systems can reconstruct types, known as type inference, for a given program without the need to be provided with explicit type annotations from programmers.

Modern type systems are often equipped with a range of features, and able to provide different kinds of guarantees according to the underlying theory. Most type systems are not designed to handle complex invariants and properties of data or program, but provide guarantees on the set of possible values exhibited by a variable, denoted by its type. Type systems with stronger guarantees, such as dependent types, come with costs. In a dependent type system it is usually impossible to provide a general type inference algorithm due to undecidability [12], and hence some type annotations are necessary. Moreover, proofs need to be supplied by programmers which can be tedious. Another approach, refinement types [17], is built upon an existing type system and enhances the expressiveness by allowing predicates, usually in the form of logical formulas, to be placed on types. In recent work [48], refinement type systems use SMT solvers to check for satisfiability of logical formulas, which saves programmers from writing tedious

manual proofs.

To reason about concurrent and distributed programs, researchers develop models of concurrent computing, of which the most notable models are shared memory and message passing. In shared memory models, components communicate by accessing a shared piece of memory. Practically, this is how different cores of the CPU, or different threads in a process operate. The model may seem simple, but complication arises when dealing with out-of-order execution, a common technique of optimisation on both architectural and software levels. On the other hand, the message passing approach models processes that communicate in the form of messages. This model reflects usages in distributed programming where high-level components exchange messages specified by protocols. The structured exchange of messages can be further described by session types [22], a type discipline which provides guarantees on communication, such as deadlock freedom, session fidelity and the absence of communication mismatch.

We motivate this research from the need to combine session types and refinement types, two disciplines that seem orthogonal, into a stronger theory that provides more expressiveness in specifying protocols. Data dependencies in communication protocols are very common, but simple session types are not expressive enough for these data dependencies. On the other hand, refinement types focus on single components, where interaction with other components may not be expressible in a straightforward way. Combining the two theories together allows the safety properties of communication from the session type side to work with the logical predicates from refinement type side, leading to this project of refinement session types.

1.2 Objectives

The objective of this project is to provide an end-to-end solution for verifying communication protocols **statically**. We use the protocol description language Scribble [51], originating from the multiparty session type (MPST) theory [23], with extensions to allow protocol refinements to be expressed. We then use code generation to provide developers with APIs, so that they can use the generated APIs to implement the communication protocols. In the end, we provide a way to verify the correctness of implementation statically, so that there is no need to perform runtime checks with regards to refinements or channel linearity. This involves a theory of refinement types along with an implementation of the type systems.

1.3 Contributions and Report Structure

We present an end-to-end solution for static verification of refined communication protocols. The related work [36] uses runtime checks for validating the refinements and checking channel linearity, whereas we present **a static approach**. To the best knowledge of the author, this is the first work on static verification of multiparty protocols with refinement, by combining multiparty session types and refinement types.

We present the workflow of the end-to-end solution in Figure 1.1.

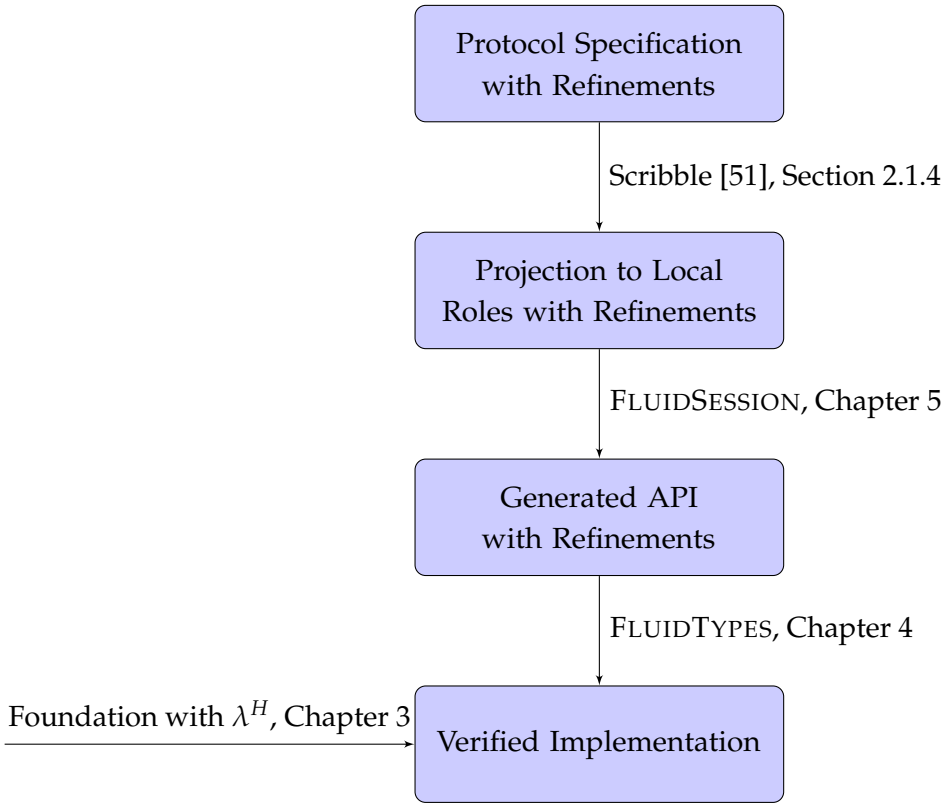

Figure 1.1: Workflow for Verified Implementation of Refined Protocols

In Chapter 2, we introduce the type discipline of session types and refinement types respectively, including related work on theory and implementation. In particular, we introduce a prior work on combining the two theories.

In Chapter 3, we introduce a variant of λ-calculus with refinement types called λ^H. We explain the typing judgements in detail and give proofs of the type safety properties. Detailed lemma and proofs in this section can be found in Appendix B.1. This chapter provides the theoretical basis for the implementation of FLUIDTYPES.

In Chapter 4, we introduce an implementation of λ^H in F# as a typechecker extension, which we call FLUIDTYPES. The implementation provides a way for programmers to annotate F# code with refinement types and check whether the refined program is well-typed. This library is not limited to checking implementations for communication protocols, but works for general F# programs. The library is publicly available on `https://github.com/fangyi-zhou/FluidTypes`.

In Chapter 5, we introduce a way to integrate FLUIDTYPES with the Scribble toolchain for multiparty session types. We present a handler style of API generation for communication protocols specified in Scribble to F#, taking refinements into account. The generated code allows developers to implement the protocol in a correct-by-construction way, guaranteed by types of the APIs. Moreover, the refinements can be checked by the library FLUIDTYPES statically, strengthening the guarantees. We have pub-

lished the code generator for multiparty protocols at `https://github.com/fangyi-zhou/ScribbleCodeGen`.

In Chapter 6, we evaluate our work through case studies of refined protocols. We demonstrate how to implement refined protocols with our work, and how FLUIDTYPES is able to verify implementations statically with respect to refinements. We also discuss limitations on expressiveness of refinements.

Finally, we summarise our work in Chapter 7 and propose possible extensions and future works on this project.

This project was presented as *Fluid Types: Statically Verified Distributed Protocols with Refinements* at 11th Workshop on Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES 2019) [52], and in the *Type My Morning* seminar series at Facebook London.

In addition to the project itself, the author has made contributions to various F# open source libraries during the project period, including feature improvements and bug fixes. The full list of contribution is listed in Section 7.1.1.

Chapter 2

Background

In this chapter, we introduce the typing discipline of session types and refinement types, which are the two important theories involved in this project.

2.1 Session Types

Many studies are done on modelling concurrency, especially on how concurrent processes communicate with each other. Message passing and shared memory are two major abstractions of communication techniques. In the shared memory model, processes have access to a shared piece of memory, analogous to different cores in CPU having access to the same physical memory. This model may seem simple, but the complication arises from common optimisations of out-of-order execution [1]. In the message passing model, processes are linked by communication channels and communicate by sending messages in the channels, analogous to a cluster of servers communicating via network in the setup of distributed systems. We focus on the latter model in this section.

Message passing can be modelled by the process algebra π-calculus originally proposed by Milner [34]. Processes can form channels identified by *names*, and can pass names via the channels. We introduce an asynchronous variant of π-calculus in Section 2.1.1. Session types provide a typing discipline for processes and channels modelled in the π-calculus. Initial work on session types [22] only supports binary sessions on channels. Honda et al. [23, 24] extend session types from binary to multiparty, enabling the theory to be applied to a vast range of real world protocols. Well-typed processes are guaranteed to have session fidelity, no type mismatch, no deadlocks and no protocol violations during the communicating sessions. With the use of session types, distributed systems and network communications are more reliable.

Session types are implemented and used in various popular programming languages of different paradigms, including C [39], Erlang [16], Java [18, 26, 27], Python [35], OCaml [28], etc. Different approaches are used to implement them according to the language design and features available for the programming language. For example, in some languages static type checking is used to check whether sessions are correctly

implemented, where others may require dynamic approaches to check type information at runtime.

2.1.1 Asynchronous π-calculus

The π-calculus provides a model of concurrent computation. Basic components, known as processes, run in parallel and communicate via names. The asynchronous π-calculus is a simple, concise, yet powerful model. It is able to encode λ-calculus, hence is Turing complete [33]. The original asynchronous π-calculus was first presented in [7, 21]. Many variants of the π-calculus exist in the literature, here we present a variant as shown in [50].

$P, Q ::=$	Processes
$\mathbf{0}$	Nil Process
$\mid \overline{u}\langle v \rangle$	Output
$\mid u(x).P$	Input
$\mid P \mid Q$	Parallel Composition
$\mid\ !P$	Replication
$\mid (\nu\, a)\, P$	Scope Restriction
$u, v ::= a$	Names
$\mid x$	Variables

Figure 2.1: Processes in Asynchronous π-calculus

We define the syntax of processes in the asynchronous π-calculus in Figure 2.1.

- $\mathbf{0}$ is the nil process, which represents a process with no action.
- $\overline{u}\langle v \rangle$ is an output process that will send v on u.
- $u(x).P$ is an input process. On receiving a message from u it can carry on executing P, with x substituted with the message received.
- $P \mid Q$ represents processes composing in parallel.
- $!P$ represents a process that can replicate itself, meaning there may be infinite processes of P composing in parallel.
- $(\nu\, a)\, P$ represents the process P with a private name a, where the name a inside process P will not interfere with other names in the open world.

Notice that the output process $\overline{u}\langle v \rangle$ does not have a continuation process P. This is due to the asynchronous nature of the calculus.

		P	$\equiv$	P	Reflexivity
$P \equiv Q$	$\Longrightarrow$	Q	$\equiv$	P	Symmetry
$P \equiv R \wedge R \equiv Q$	$\Longrightarrow$	P	$\equiv$	Q	Transitivity
$P \equiv Q$	$\Longrightarrow$	$(\nu\, a)\, P$	$\equiv$	$(\nu\, a)\, Q$	Congruence of Restriction
$P \equiv Q$	$\Longrightarrow$	$P \mid R$	$\equiv$	$Q \mid R$	Congruence of Parallel
$P \equiv Q$	$\Longrightarrow$	$\overline{u}\langle x \rangle.P$	$\equiv$	$\overline{u}\langle x \rangle.Q$	Congruence of Input
$P \equiv Q$	$\Longrightarrow$	$!P$	$\equiv$	$!Q$	Congruence of Replication
$P =_\alpha Q$	$\Longrightarrow$	P	$\equiv$	Q	α-equivalence
		$(P \mid Q) \mid R$	$\equiv$	$P \mid (Q \mid R)$	Associativity
		$P \mid Q$	$\equiv$	$Q \mid P$	Commutativity
		$P \mid \mathbf{0}$	$\equiv$	P	Nil
		$!P$	$\equiv$	$P \mid !P$	Replication
		$(\nu\, a)\, \mathbf{0}$	$\equiv$	$\mathbf{0}$	Restriction of Nil
		$(\nu\, a)\, (\nu\, b)\, P$	$\equiv$	$(\nu\, b)\, (\nu\, a)\, P$	Restriction of Restriction
$a \notin \mathrm{fn}(P)$	$\Longrightarrow$	$P \mid (\nu\, a)\, Q$	$\equiv$	$(\nu\, a)\, (P \mid Q)$	Restriction of Parallel

Table 2.1: Structural Congruence for Asynchronous π-calculus

An important concept in the π-calculus is the structural congruence relation. Structurally congruent processes are not distinguishable. We define this relation in Table 2.1.

$$\frac{}{\overline{a}\langle v \rangle \mid a(x).P \rightarrow P[v/x]}\ \textsc{Comm}$$

The key rule of the operational semantics of asynchronous π-calculus is (COMM), where an input process and an output process, composing together in parallel, reduce when the names are the same. This models the communication between the input process and the output process on a channel. The remaining process is the continuation process from the input process, with the variable substituted by the message received.

2.1.2 Binary Session Types

A *session* is a sequence of actions on a channel. In the typing discipline of session types, we give types to communication channels. Names are distinguished between two use cases, as session names or shared names. A shared name is public and it acts as a communication channel for unrestricted number of parties. A session name is private to communicating parties and acts as a *linear channel* where communication is restricted to the two parties.

We introduce a session calculus with different syntactic constructs for linear and shared names for processes, to address the difference between session and shared names.

$P, Q ::=$	Processes
$\mathbf{0}$	Nil Process
$\mid \overline{u}\langle e \rangle.P$	Output on Shared names
$\mid u(x).P$	Input on Shared names
$\mid \overline{k}\langle e \rangle.P$	Output on Session names
$\mid k(x).P$	Input on Session names
$\mid k \triangleleft \{l_1 : P_1 \,[]\, \cdots \,[]\, l_n : P_n\}$	Branches on Session names
$\mid k \triangleright l.P$	Selection on Session names
$\mid P \mid Q$	Parallel Composition
$\mid (\nu\, a)\, P$	Scope Restriction on Shared names
$\mid (\nu\, k)\, P$	Scope Restriction on Session names
$\mid$ `if` e `then` P `else` Q	Conditionals
$\mid$ `def` D `in` P	Recursive Definitions
$\mid X\langle \tilde{e} \rangle$	Recursive Calls
$u ::= a \mid x$	Shared Names and Variables
$k ::= s \mid x$	Session Names and Variables
$e ::= v \mid e \oplus e \mid \neg e$	Expressions
$\oplus ::= \wedge \mid \vee$	Binary operators
$v ::=$ `true` $\mid$ `false` $\mid s \mid a$	Values
$D ::= X(\tilde{x}) = P$	Recursion Declarations

Figure 2.2: Processes in Session Calculus

We define the syntax of the processes in Figure 2.2. This calculus is an extended version of Figure 2.1, with additional constructs for:

- Syntactic distinction between shared names and session names. This is represented by two variants of input and output processes.
- Recursive definitions and calls . This is represented by `def` D `in` P and $X\langle \tilde{e} \rangle$ processes. $X\langle \tilde{e} \rangle$ is expanded into the process in the definition, with usual function call behaviour.
- Expressions with boolean constants and conditional processes. The process `if` e `then` P `else` Q reduces to P if e evaluated to `true` and Q otherwise.
- Branching and selection processes. A process $k \triangleleft \{l_1 : P_1 \,[]\, \cdots \,[]\, l_n : P_n\}$ can provide one or more branches with distinct labels on a session name. The branch can be selected by the process $k \triangleright l.P$.
- Output processes $\overline{u}\langle e \rangle.P$ have continuation processes. After a successful commu-

nication, it continues with process P. This means that this session calculus in synchronous.

The syntax of session types are defined in Figure 2.3. The syntax of session types and processes share a large similarity. In particular, input processes correspond to *send* types, and output processes to *receive* types, etc. Hence in many cases, session types can be assigned to session variables and names in a syntax directed fashion.

$S ::=$		Sort
	$\texttt{bool}$	Boolean
$T, U ::=$		Type
	$![S]; T$	Value Send
	$\mid ?[S]; T$	Value Receive
	$\mid ![U]; T$	Type Send
	$\mid ?[U]; T$	Type Receive
	$\mid \&\{l_1 : T_1, \cdots, l_n : T_n\}$	Branches
	$\mid \oplus \{l_1 : T_1, \cdots, l_n : T_n\}$	Selection
	$\mid t$	Type Variable
	$\mid \mu t.T$	Recursive Type
	$\mid \texttt{end}$	Termination

Figure 2.3: Syntax of Session Types

A key concept in session types is *duality*. Process reduction occurs when communication takes place, that is when a message is exchanged on a name. There should be two processes at two ends of the channel, a "sending" side and a "receiving" side, expecting dual behaviours on two ends of the channel. The two ends, identified by their names, should be typed in a *dual* form, so that we can prevent communication mismatches from happening. Following this intuition, we define $\overline{T}$, the dual of type T, in Table 2.2.

When two processes compose in parallel, all session variables in the processes should carry dual types. In this way, communication always matches, and mismatch errors or deadlocks never occur due to duality of types. A send action is always matched with a receive action, and vice versa.

In addition, the linear names must be used once and only once between in two processes composing in parallel. This ensures a *use-once* property that a message sent will be eventually received, so that there is no orphaned messages.

We described the desired guarantees and intuitions for the session types. For exact details of the typing judgements, readers can refer to [19, 22, 47].

$$
\begin{array}{rcl}
\overline{![S];T} & = & ?[S];\overline{T} \\
\overline{?[S];T} & = & ![S];\overline{T} \\
\overline{![U];T} & = & ?[U];\overline{T} \\
\overline{?[U];T} & = & ![U];\overline{T} \\
\overline{\&\{l_1 : T_1, \cdots, l_n : T_n\}} & = & \oplus\{l_1 : \overline{T}_1, \cdots, l_n : \overline{T}_n\} \\
\overline{\oplus\{l_1 : T_1, \cdots, l_n : T_n\}} & = & \&\{l_1 : \overline{T}_1, \cdots, l_n : \overline{T}_n\} \\
\overline{t} & = & t \\
\overline{\mu t.T} & = & \mu t.\overline{T} \\
\overline{\mathtt{end}} & = & \mathtt{end}
\end{array}
$$

Table 2.2: Duality of Session Types

2.1.3 Multiparty Session Types

While session types in Section 2.1.2 provide some guarantees, their expressiveness is limited to describing communication protocols between two parties. In distributed computing, there are usually more parties involved. A large multiparty protocol is not always decomposable into smaller binary session types between two communicating parties. In particular, the duality of session types does not extend to multiple parties easily.

Multiparty session types (MPST) are a major extension to the binary session types, which are able to address these limitations. The typing discipline provides more expressiveness via a view of the communication protocol from the global perspective. The global view can then be projected into local views where processes can be monitored whether they obey the communication protocol [37], and code can be generated for each local process [38], via local communicating finite state machines.

The syntax of processes in multiparty session calculus, as presented in [9], is shown in Figure 2.4. In this session calculus, message delivery is modelled in an asynchronous fashion, where messages queue up waiting to be delivered. Since multiple parties are involved in the communication, the input and output processes now carry the *participant number* of the other party, which identifies the interacting party.

Note that shaded syntax in Figure 2.4 indicates that the syntax is a runtime syntax, meaning that the syntax cannot occur in any written program, but only occurs in operational semantics.

The operational semantics for asynchronous multiparty session calculus involve message queues between processes, and include usual communication rules via channel names:

- A session name and a message queue dedicated for the session name are created after successful communication of multicast request and multicast accept processes. Each process knows its participant number in the session.

- Output of values and channels, and selection processes on a session add a message

$P, Q ::=$		Processes
	$\mathbf{0}$	Nil Process
	$\mid \overline{u}[\mathsf{p}](y).P$	Multicast Request
	$\mid u[\mathsf{p}](y).P$	Multicast Accept
	$\mid \overline{c}\langle \mathsf{p}, e \rangle.P$	Output of Values
	$\mid c(\mathsf{p}, x).P$	Input of Values
	$\mid \overline{c}\langle \langle \mathsf{p}, c' \rangle \rangle.P$	Output of Channels
	$\mid c((\mathsf{p}, y)).P$	Input of Channels
	$\mid c \triangleleft (\mathsf{p}, \{l_1 : P_1 \,[]\, \cdots \,[]\, l_n : P_n\})$	Branches
	$\mid c \triangleright (\mathsf{p}, l).P$	Selection
	$\mid P \mid Q$	Parallel Composition
	$\mid (\nu\, a)\, P$	Scope Restriction on Shared names
	$\mid (\nu\, s)\, P$	Scope Restriction on Session names
	$\mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditionals
	$\mid \texttt{def}\ D\ \texttt{in}\ P$	Recursive Definitions
	$\mid X\langle e, c \rangle$	Recursive Calls
	$\mid s : h$	Message Queue
$D ::=$	$X(x, y) = P$	Recursion Declarations
$u ::=$	$a \mid x$	Shared Names and Variables
$e ::=$	$v \mid x \mid e \oplus e \mid \neg e$	Expressions
$\oplus ::=$	$\wedge \mid \vee$	Binary operators
$v ::=$	$\texttt{true} \mid \texttt{false} \mid a$	Values
$c ::=$	$y \mid s[\mathsf{p}]$	Channels
$m ::=$		Messages in Transit
	$(\mathsf{q}, \mathsf{p}, v)$	Message of value
	$\mid (\mathsf{q}, \mathsf{p}, s[\mathsf{p'}])$	Message of channel
	$\mid (\mathsf{q}, \mathsf{p}, l)$	Message of label
$h ::=$	$h \cdot m \mid \varnothing$	Message Queue

Figure 2.4: Processes in Multiparty Session Calculus

to the message queue of that session.

- Input of values and channels, and branching processes on a session remove a message from the message queue of that session.
- Messages in the queues are tagged with participant numbers of the sending and receiving processes.

Typing for multiparty session calculus involves a global description of protocol, known as a *global type* involving message exchanges between all participants. Due to the multiparty nature, a session can have more than two participants, hence the session types are also annotated with participant number, which does not exist in binary session types, to indicate the intended target in the session. Using this idea, we define the syntax of global types and session types in Figure 2.5.

$S ::=$		Sort
	$\texttt{bool}$	Boolean
$G ::=$		Global Types
	$\mathsf{p} \to \mathsf{q} : \langle S \rangle . G$	Value Exchange
	$\mid \mathsf{p} \to \mathsf{q} : \langle T \rangle . G$	Channel Exchange
	$\mid \mathsf{p} \to \mathsf{q} : \{l_i : G_i\}_{i \in I}$	Branching
	$\mid t$	Type Variable
	$\mid \mu t . T$	Recursive Type
	$\mid \texttt{end}$	Termination
$T, U ::=$		Session Type
	$!\langle \mathsf{p}, S \rangle ; T$	Value Send
	$\mid ?(\mathsf{p}, S) ; T$	Value Receive
	$\mid !\langle \mathsf{p}, U \rangle ; T$	Channel Send
	$\mid ?(\mathsf{p}, U) ; T$	Channel Receive
	$\mid \&(\mathsf{p}, \{l_i : T_i\}_{i \in I})$	Branches
	$\mid \oplus (\mathsf{p}, \{l_i : T_i\}_{i \in I})$	Selection
	$\mid t$	Type Variable
	$\mid \mu t . T$	Recursive Type
	$\mid \texttt{end}$	Termination

Figure 2.5: Global and Session Types

Local types describe the communicating behaviour of a local role in the global protocol. The local types for processes can be projected from a global type for a given role in the protocol, as shown in Figure 2.6. With local types, each role has its own view of the communication protocol besides the global view. Practically, this means that

development of each participant in the protocol can be done in isolation – as long as the projection is correct, composing different components developed in isolation gives correct behaviour as described in the global protocol. We explain more detail in Section 2.1.4 with an example of how a protocol can be described in the Scribble language with API generation for different endpoints.

$$(\mathtt{p} \to \mathtt{p'} : \langle U \rangle . G') \upharpoonright \mathtt{q} = \begin{cases} !\langle \mathtt{p'}, U \rangle ; (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p} \\ ?(\mathtt{p}, U) ; (G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p'} \\ (G' \upharpoonright \mathtt{q}) & \text{otherwise} \end{cases}$$

$$(\mathtt{p} \to \mathtt{p'} : \{l_i : G_i\}_{i \in I}) \upharpoonright \mathtt{q} = \begin{cases} \oplus(\mathtt{p'}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}) & \text{if } \mathtt{q} = \mathtt{p} \\ \&(\mathtt{p}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i \in I}) & \text{if } \mathtt{q} = \mathtt{p'} \\ (G_{i_0} \upharpoonright \mathtt{q}) & \text{where } i_0 \in I \text{ if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \neq \mathtt{p'} \\ & \text{and } G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q}, \text{ for all } i, j \in I \end{cases}$$

$$(\mu t.G) \upharpoonright \mathtt{q} = \begin{cases} \mu t.(G \upharpoonright \mathtt{q}) & \text{if } G \upharpoonright \mathtt{q} \neq t \\ \mathtt{end} & \text{otherwise} \end{cases}$$

$$(t) \upharpoonright \mathtt{q} = t$$

$$(\mathtt{end}) \upharpoonright \mathtt{q} = \mathtt{end}$$

Figure 2.6: Projection from Global Types to Session Types

Multiparty session types are able to provide guarantees on communication, including:

- **Communication Safety:** There are no mismatch in expected type and the actual type when sending and receiving messages on a channel.
- **Protocol Fidelity:** Interaction between processes will follow the global protocol.
- **Progress:** Messages sent by processes will be eventually received, and processes waiting for messages will eventually receive messages. This also implies that there will be no orphaned messages.

2.1.4 Scribble and Endpoint API Generation

Scribble [51] is a protocol description language that incorporates multiparty session types, applying the theory into practical programming. Communicating parties, known as *roles*, exchange typed structured messages for interaction. Each message has a label and a payload signature, where the label distinguishes different kinds of the messages in the protocol and the payload signature specifies the *types* of the payload included in the message.

An example protocol described in Scribble is in Figure 2.7. The protocol describes a server capable of performing addition operation. After sending a `HELLO` message to the

server, the client can choose to send two integers in a ADD message and receive the result in a RES message, and repeat the protocol. Alternatively, the client can choose to send a BYE message and terminate the protocol after receiving a BYE message from the server.

```
1  global protocol Adder(role C, role S) {
2      HELLO(int) from C to S;
3      choice at C {
4          ADD(int, int) from C to S;
5          RES(int) from S to C;
6          do Adder(C, S);
7      } or {
8          BYE() from C to S;
9          BYE() from S to C;
10     }
11 }
```

Figure 2.7: Adder Protocol

The global protocol can be projected to local roles so that each role can use the projected protocol to generate code for the structured communication for that given role, independent of other roles. A *communicating finite state machine* (CFSM) describes local behaviour of the given role, where state transitions are actions of communication, i.e. sending messages to or receiving messages from other roles.

We take Java as an example for the code generation process. Each state in the CFSM is be converted into a class, with outgoing transitions converted into its methods. The methods contain communication primitives for sending and receiving messages, and return the next state in the CFSM. In this way, the programmer can use a chain of methods starting from the initial state to reach a terminating state where there is no outgoing transitions. It is important that each state object must be used once and only once, to ensure the CFSM has correct state transitions.

Implementations following the generated APIs can benefit from the guarantees of MPST, i.e. free from communication errors, session fidelity and progress.

To illustrate, Figure 2.8 shows the finite state machine for role C (client), obtained via Scribble from the Adder protocol in Figure 2.7. At state 2, the client makes an choice on what message to send to the server, either a BYE message to terminate, or an ADD message for computation. We show the generated code in Java for the state in Figure 2.9.

2.1.5 F# Session Type Provider

F# is an open source programming language, originally developed by Microsoft. It is a member of ML language family, and shares a similar syntax to OCaml. Type provider [40] is a language feature of the F# language for compile time type generation. Common usage of this feature includes generating type-safe APIs for structured data sources, e.g. for a JSON file or a SQL database. This is done by invoking the type provider during compilation to obtain type information from data schemas or remote

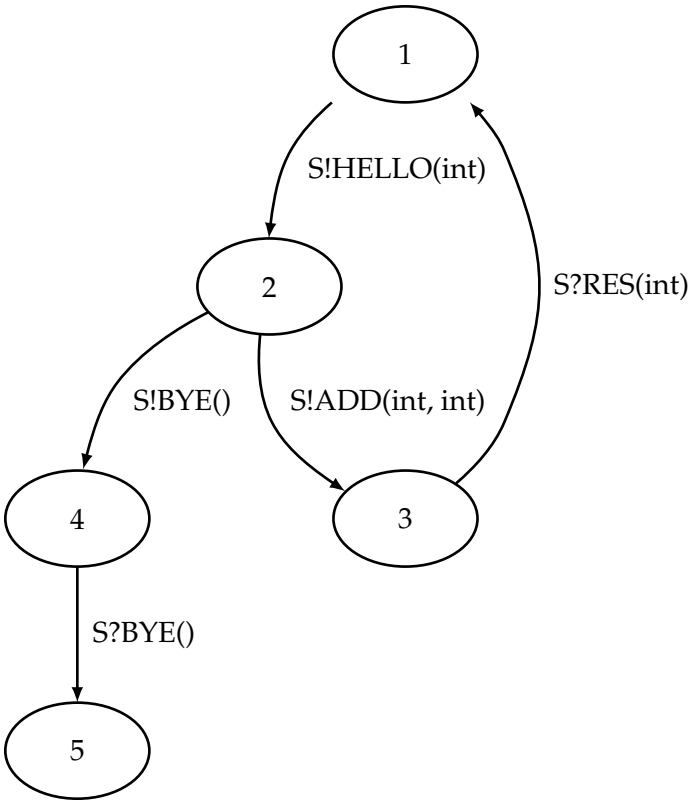

Figure 2.8: Finite State Machine for role C for Protocol in Figure 2.7

```
public Adder_C_3 send(S role, ADD op, int arg0, int arg1) {
    super.writeScribMessage(role, Adder.ADD, arg0, arg1);
    return new Adder_C_3(this.se, true);
}

public Adder_C_4 send(S role, BYE op) {
    super.writeScribMessage(role, Adder.BYE);
    return new Adder_C_4(this.se, true);
}
```

Figure 2.9: Generated Java API for State 2, Role C (truncated)

resources, and provide typed interaction of the information in the program. Developers do not need to manually specify type definitions, and they can also benefit from typed access to resources.

Neykova et al. [36] implement multiparty session types in F# using type providers. This is an innovative use of type providers as a meta-programming technique. The communication protocol is considered as the data source and types for the local endpoints can be generated via the type provider during the compilation. This is done by projecting the global protocol to local endpoints during the compilation using Scribble, and then generate typed APIs for the local endpoint, in a similar way to Java code generation.

An important aspect of this work includes an extension of multiparty session types with refinements. We explain this extension in more detail in Section 2.2.4.

2.2 Refinement Types

Refinement type systems build upon the existing type system of a programming language. Refinement types offer the ability to add predicates on existing data types, restricting the values corresponding to the underlying data types [17]. Using refinements in the type signature, one can encode pre- and post-conditions of a function easily via refining the input and output types respectively.

Refinement types are implemented for many general purpose programming languages, including F# [3], Scala [44] and Haskell [48]. In many implemtations, the refinement predicates are transformed into a verification condition, which is then discharged via an external SMT solver. After the conditions are verified, the refinements do not need to be carried into the runtime, since they have been verified statically. Therefore, they can be safety erased into the underlying type system. In this way, programs can be made more reliable without performance penalty.

2.2.1 Liquid Types

Rondon et al. propose *Logically Qualified Data Types* in [43], abbreviated to *Liquid Types*. The predicates used in the refinement are limited to a decidable logic, so that refinement type inference is decidable. The type system formalisation involves a semantic subtyping relation, decided by an external SMT solver. Terms and typing contexts are encoded into SMT logic, and subtyping is modelled as implications in the logic, which are validated by a SMT solver.

Suppose `int` is the bast type for integers. Natural numbers, which are non-negative integers, can be defined as follows in liquid types:

$$\{\nu : \texttt{int} \mid \nu \geq 0\}$$

where the variable ν stands for the members of that type.

Function types in liquid types are parameterised by variables, so that the variables can be later used in the predicate as a refinement. This is a form of dependent functions.

For example, the function `max` taking the greater of two integers can take the following type, which specifies that the output must be greater than or equal to both input arguments. Whilst this may not be the most precise specification of the output, this signature of the `max` function more precisely than a function type without refinements:

$$(x : \texttt{int}) \to (y : \texttt{int}) \to \{\nu : \texttt{int} \mid \nu \geq x \wedge \nu \geq y\}$$

The refinement predicates are restricted to a set of decidable logic of equality, uninterpreted functions and linear arithmetic (EUFA). Terms in the refinement predicate and the typing context are encoded in the logic conservatively. The subtyping relation is decided by the validity of implication of the encoded types and typing contexts, as described by the following typing rule ($[\![e]\!]$ represents the encoding of e in the logic and B represents a base type). We use a similar formalisation of typing judgements in our work, which we explain in detail in Section 3.4.

$$\frac{\mathsf{Valid}([\![\Gamma]\!] \wedge [\![e_1]\!] \implies [\![e_2]\!])}{\Gamma \vdash \{\nu : B \mid e_1\} <: \{\nu : B \mid e_2\}}$$

2.2.2 F7: Refinement Types for F#

F7 [20] is an extension to the F# language, a dialect of ML, with refinement types developed by Microsoft Research. Currently, there is no active development of F7 after its release. In F7, the theory includes formalisation of a calculus with refinements and concurrency, with features such as algebraic data types, dependent function and pair types and π-calculus style message passing communication primitives.

The logical predicates allowed to refine the data types include first-order logical operators. An SMT solver (Z3 [10]) is used during type checking. The work produces a partial inference algorithm for the type system, hence some type need to be annotated manually. In addition, due to the incompleteness of Z3 prover, some valid first order formulas are not provable. In contrast, liquid types are always decidable, but the logical operators permitted are more restricted.

Bengtson et al. [3] apply this refinement calculus to cryptography scenarios. In this formalisation, processes can communicate with typed channels. The typed channel only allows values of the specified type to be passed. In contrast, session types are more expressive, since values of different types may be transmitted via channels, with their order and direction of transfer specified.

Bhargaven et al. [5] demonstrate a technique for protocol synthesis and verification of multiparty session types. A protocol specification is compiled into protocol implementation files and a typed interface, which can be checked for correctness by F7 with session types encoded as logical pre- and post-conditions.

2.2.3 F*

F* [31] is a programming language developed by Microsoft Research and INRIA for program verification. The type system of F* subsumes previous work of F7. The

language has a type system with a rich set of features, including dependent types, refinement types and a weakest precondition calculus. F* plays a key role in Project Everest [4, 42] for a fully verified HTTPS stack.

Swarmy et al. [45] provide an example for verifying multiparty sessions using F*, applying multiparty session type theory in [11, 23]. With support of affine types in F*, the technique can produce more precise specifications with less code annotations compared to previous work in [5].

2.2.4 F# Session Type Provider

Neykova et al. [36] extend multiparty session types with refinements. The protocol specification language Scribble is extended to include refinements. For example, the Adder protocol as shown in Figure 2.7 can be refined as shown in Figure 2.10. The refined protocol imposes restrictions on the values exchanged in messages. Essentially, the Adder only works with positive integers rather than all integers.

```
1  global protocol Adder(role C, role S) {
2      HELLO(int) from C to S;
3      choice at C {
4          ADD(x:int, y:int) from C to S; @"x > 0 && y > 0"
5          RES(v:int) from S to C; @"v > 0"
6          do Adder(C, S);
7      } or {
8          BYE() from C to S;
9          BYE() from S to C;
10     }
11 }
```

Figure 2.10: Refined Adder Protocol

During the code generation phase, the refinement conditions are checked by Scribble for well-formedness, including the following:

- **Variable Knowledge:** The refinement on a message may not refer to a variable not known by sending endpoint. This ensures that refinements can be checked during runtime for the sending endpoint.
- **Refinement Satisfiability:** Every refinement must be satisfiable by some assignments of variables. This ensures that there are no unreachable parts of the protocol.
- **Refinement Progress:** Whenever there is a choice in the protocol, there must be at least one branch whose refinements are satisfiable, under all assignments of variables. This ensures that there is no case that a protocol gets stuck with no feasible branches due to unsatisfied refinements.

During the runtime, refinements are encoded as inline assertions. At each endpoint, a cache is used to store variables, so that refinements can be checked when needed. During

execution, unsatisfied refinements lead to runtime expections in this implementation. The runtime refinement checks incur an overhead during execution.

We motivate our work from this existing work. We develop a static approach of refinement session types to address the limitations.

Chapter 3

λ^H: A Simple Refinement Calculus

In this chapter, we define a variant of the λ-calculus with refinement types. We build upon the type system of the simply typed λ-calculus (hereafter STLC) with integer and boolean literals, and extend it to include refinements on the type level, which impose constraints on the members of a given type.

The name λ^H originally comes from [29]. The refinement typed variant comes from [43]. The form of λ^H presented in this chapter is adopted from these previous work. We give syntax, semantics, bidirectional typing judgements, and type safety theorems in this chapter.

λ^H provides the theoretical basis for the implementation of FLUIDTYPES in F#. We explain the implementation in detail in Chapter 4.

3.1 Syntax

We define the syntax of terms and types in λ^H in Figure 3.1. The terms M and types τ are defined in a mutually recursive way, since a type can occur in a term in a type annotated term $(M : \tau)$, and a term can occur in a type in a base type $(\{\nu : b \mid M\})$.

We define terms in two syntactic categories to support bidirectional typing [41], which we discuss in detail in Section 3.4. The two syntactic categories for all terms are $M^\star$ for synthesisable terms and M for checkable terms. The former category has a typing judgement for synthesis, meaning a type can be reconstructed from a given context, whereas the latter has a typing judgement for checking, meaning a type needs to be provided to check whether the term is well-typed under a given context.

The terms M in λ^H include the usual terms in λ-calculus, namely variables (x), abstractions $(\lambda x.M)$ and applications $(M^\star M)$. In addition, there are syntactic constructs for:

- Conditional expressions (`if` M `then` M `else` M), a common extension to the λ-calculus;
- Type-annotated terms $(M : \tau)$, which allows a checkable term to carry an explicit type annotation to become synthesisable;

$$
\begin{array}{rll}
M^\star ::= & & \text{Synthesisable Term} \\
& x & \text{Variable} \\
\mid & c & \text{Constant} \\
\mid & M^\star M & \text{Application} \\
\mid & M : \tau & \text{Type-annotated Term} \\
M ::= & & \text{Checkable Term} \\
& \lambda x.M & \text{Abstraction} \\
\mid & \texttt{if } M \texttt{ then } M \texttt{ else } M & \text{Conditional} \\
\mid & M^\star & \text{Synthesisable Term} \\
b ::= & \texttt{int} \mid \texttt{bool} & \text{Base Type} \\
\tau ::= & & \text{Refinement Type} \\
& \{\nu : b \mid M\} & \text{Refined Base Type} \\
\mid & (x : \tau) \to \tau & \text{Function type}
\end{array}
$$

Figure 3.1: Syntax of λ^H

- Constants (c). In this cases, they consist of integer literals (e.g. `0`, `1`, `2` ...), boolean literals (i.e. `true`, `false`), and operators for integers and booleans (e.g. `(&&)`, `(+)`, `(=)`). We write operators in infix forms where appropriate for clarity.

For types, we first have the base types for integers and booleans, which corresponds to the integer and boolean literals in terms. A refinement type τ is either a refined base type ($\{\nu : b \mid M\}$) or a function type ($(x : \tau_1) \to \tau_2$). A refined base type $\{\nu : b \mid M\}$ is in a form similar to a mathematical set notation. It consists of a base type b, indicating the underlying type of the value, and a term M, acting as a predicate on the type. The predicate term M specifies a constraint on the values of the refined base type, such that a member ν needs to satisfy the predicate term M. This is the reason why this typing discipline is known as refinement types, as the predicate refines values in the underlying base types. As a consequence, M is a term of a boolean type (later explained in typing rules), and it may optionally contain the special variable ν, which stands for the members of the refined type. For example, one can represent positive integers with type $\{\nu : \texttt{int} \mid \nu > 0\}$, meaning all its members ν should satisfy the predicate $\nu > 0$.

In the simply typed λ-calculus, a function type is usually of form $t_1 \to t_2$, where t_1 and t_2 are types. In our syntax, a function type is of form $(x : \tau_1) \to \tau_2$, where the argument type τ_1 is named with a variable. Moreover, the variable x can occur in the result type τ_2 in the predicates. In other words, this function type represents dependent function space, or is sometimes written in an alternative form of $\Pi_{x:\tau_1} \tau_2(x)$, common in presentations of Martin-Löf type theory [30].

This definition of types can also represent those types without refinements. For example, the type for all integers, i.e. `int` in simple type system, corresponds to $\{\nu : \texttt{int} \mid \texttt{true}\}$, since all integers satisfy the predicate `true`. In fact, we use base type

b to abbreviate $\{\nu : b \mid \texttt{true}\}$ for simplicity, where there is no ambiguity. In cases where a function $(x : \tau_1) \to \tau_2$ is not dependent, such that the variable x does not occur τ_2, we use the abbreviation $\tau_1 \to \tau_2$.

For typing, we define two contexts for typing judgements, the typing context Γ for storing types for variables and the predicate context Δ for storing refinement predicates. We explain their roles in typing later in Section 3.4.

$\Gamma ::=$	$\varnothing \mid \Gamma, x : \tau$	Typing Context
$\Delta ::=$	$\varnothing \mid \Delta, M$	Predicate Context

Figure 3.2: Definition of Contexts

We define free variables for terms and types in Definition 3.1.1. The definition of free variables does not deviate too much from the λ-calculus. In an abstraction $\lambda x.M$, the variable x is bound in M. For types, in a base type $\{\nu : b \mid M\}$, the variable ν is bound in M, as it is a special variable representing the value exhibited by the type; in function type $(x : \tau_1) \to \tau_2$, the variable x is bound in τ_2. We define substitutions for terms, types and typing context in Definition 3.1.2, in the usual way that only free variables are substituted, and that bound variables are not affected by substitution.

Definition 3.1.1 (Free Variables)**.** *Free variables of a term are defined as follows:*

$$\begin{aligned}
\mathrm{fv}(x) &= \{x\} \\
\mathrm{fv}(c) &= \varnothing \\
\mathrm{fv}(\lambda x.M) &= \mathrm{fv}(M) \setminus \{x\} \\
\mathrm{fv}(M_1 M_2) &= \mathrm{fv}(M_1) \cup \mathrm{fv}(M_2) \\
\mathrm{fv}(\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3) &= \mathrm{fv}(M_1) \cup \mathrm{fv}(M_2) \cup \mathrm{fv}(M_3) \\
\mathrm{fv}(M : \tau) &= \mathrm{fv}(M) \cup \mathrm{fv}(\tau)
\end{aligned}$$

Free variables of a type are defined as follows:

$$\begin{aligned}
\mathrm{fv}(\{\nu : b \mid M\}) &= \mathrm{fv}(M) \setminus \{\nu\} \\
\mathrm{fv}((x : \tau_1) \to \tau_2) &= \mathrm{fv}(\tau_1) \cup (\mathrm{fv}(\tau_2) \setminus \{x\})
\end{aligned}$$

Definition 3.1.2 (Substitution)**.** *Substitution on terms is defined as follows.*

$$\begin{aligned}
x[M/x] &= M \\
y[M/x] &= y && (x \neq y) \\
c[M/x] &= c \\
(\lambda y.N)[M/x] &= \lambda y.(N[M/x]) && (x \neq y) \\
(M_1 M_2)[M/x] &= M_1[M/x] M_2[M/x] \\
(\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3)[M/x] &= \texttt{if } M_1[M/x] \texttt{ then } M_2[M/x] \texttt{ else } M_3[M/x] \\
(N : \tau)[M/x] &= N[M/x] : \tau[M/x]
\end{aligned}$$

Substitution on types is defined as follows.

$$(\{v : b \mid N\})[M/x] = \{v : b \mid N[M/x]\}$$
$$((y : \tau) \to \sigma)[M/x] = ((y : \tau[M/x]) \to \sigma[M/x]) \qquad (x \neq y)$$

Substitution on typing contexts is defined as follows.

$$(\varnothing)[M/x] = \varnothing$$
$$(\Gamma, y : \tau)[M/x] = \begin{cases} \Gamma[M/x], y : \tau[M/x] & \text{if } x \neq y \\ \Gamma[M/x] & \text{if } x = y \end{cases}$$

Substitution on predicate contexts is defined as follows.

$$(\varnothing)[M/x] = \varnothing$$
$$(\Delta, N)[M/x] = \Delta[M/x], N[M/x]$$

3.2 Operational Semantics

We define operational semantics for λ^H in a call by value style. We add additional rules for additional constructs that do not occur in the λ-calculus.

We first define values of λ^H in Definition 3.2.1.

Definition 3.2.1. *Values in λ^H are defined as follows:*

$v ::=$	*Values*
c	*Constant*
$\mid \lambda x.M$	*Abstractions*
$\mid v : \tau$	*Type-Annotated Values*

Constants and abstractions are always values. In addition, a value can be type-annotated and the annotated term is still a value.

We then define the operational semantics in Figure 3.3.

$$\frac{}{(\lambda x.M_1 : ((x : \tau_1) \to \tau_2))v_2 \to M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]}\ \text{RED-App-Anno}$$

$$\frac{}{(\texttt{if true then } M_2 \texttt{ else } M_3) \to M_2}\ \text{RED-If-True}$$

$$\frac{}{(\texttt{if false then } M_2 \texttt{ else } M_3) \to M_3}\ \text{RED-If-False}$$

$$\frac{M \to M'}{C[M] \to C[M']}\ \text{RED-Ctx}$$

$$C ::= \cdot M \mid v \cdot \mid \texttt{if } \cdot \texttt{ then } M \texttt{ else } M \mid \cdot : \tau$$

Figure 3.3: Reduction Rules for λ^H

In the λ-calculus, we have a reduction rule as follows:

$$\frac{}{(\lambda x.M_1)v_2 \to M_1[v_2/x]}$$

However, in λ^H, we have rule (RED-APP-ANNO) instead. This may seem unusual, but the rule is set in a way such that syntactic categories of terms are preserved. In our syntax, applications are of form $M^\star M$, and abstractions $\lambda x.M$ do not belong to $M^\star$. In order for an abstraction to be in $M^\star$, a type annotation is necessary, since a type annotated term $(M : \tau)$ is in $M^\star$. We future note that M_1 is in M, and hence it remains in M after substitution. Since an application is in $M^\star$, the result after substitution is annotated to preserve the syntactic category. Similarly, we substitute x with $v_2 : \tau_1$ instead of with v_2.

(RED-IF-FALSE) and (RED-IF-FALSE) define how conditionals are reduced when the predicate term is a boolean literal. The correspondent branch is picked according to the literal.

(RED-CTX) defines the contextual closure of the base rules.

We define the multistep reduction relation $\to^*$ as the reflexive, transitive closure of the reduction relation $\to$.

3.3 Erasing Refinements

λ^H can be erased into the the λ-calculus with booleans. We define the erasure function in Definition 3.3.1 to convert type, terms and typing contexts from λ^H to the λ-calculus.

Definition 3.3.1. *The erasure of terms is defined as follows.*

$$\begin{aligned}
\text{erase}(x) &= x \\
\text{erase}(c) &= c \\
\text{erase}(\lambda x.M) &= \lambda x.\,\text{erase}(M) \\
\text{erase}(M_1 M_2) &= \text{erase}(M_1)\,\text{erase}(M_2) \\
\text{erase}(\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3) &= \texttt{if } \text{erase}(M_1) \texttt{ then } \text{erase}(M_2) \texttt{ else } \text{erase}(M_3) \\
\text{erase}(M : \tau) &= \text{erase}(M)
\end{aligned}$$

The erasure of types is defined as follows.

$$\begin{aligned}
\text{erase}(\{v : b \mid M\}) &= b \\
\text{erase}((x : \tau_1) \to \tau_2) &= \text{erase}(\tau_1) \to \text{erase}(\tau_2)
\end{aligned}$$

The erasure of typing context is defined as follows.

$$\begin{aligned}
\text{erase}(\varnothing) &= \varnothing \\
\text{erase}(\Gamma, x : \tau) &= \text{erase}(\Gamma), x : \text{erase}(\tau)
\end{aligned}$$

The concept of erasure is important, as the type system of λ^H builds upon the underlying type system of the simply typed λ-calculus. When it is necessary to use the underlying type system, we erase λ^H down to the λ-calculus. Moreover, if well-typed terms in λ^H are also well-typed after erasure, we can first typecheck the term in λ^H, then erase it to the λ-calculus for the actual execution. This is very useful for the implementation, as the runtime does not need to be modified to adopt to type system changes. We revisit erasure in Section 3.4.4 and establish connections between two type systems.

Theorem 3.3.2. *If* $M \to M'$, *then* $\text{erase}(M) \to \text{erase}(M')$

Proof. By induction on the reduction relation $\to$ of λ^H. □

3.4 Typing

In this section, we explain the bidirectional type system for λ^H, based on an existing type system of λ-calculus. We do not go into details of the base type system of the simply typed λ-calculus, but we include the typing judgements in Appendix A.1 for reference. Typing judgements defined in this chapter are summarised in Table 3.1.

$\vdash \Gamma; \Delta$	Well-formedness of Context
$\Gamma; \Delta \vdash \tau$	Well-formedness of Type
$\Gamma; \Delta \vdash \tau <: \sigma$	Subtyping
$\Gamma; \Delta \vdash M^\star \Rightarrow \tau$	Type Synthesis
$\Gamma; \Delta \vdash M \Leftarrow \tau$	Type Checking
$\Gamma; \Delta \vdash_{\text{erase}} M : \tau$	Typing under erasure

Table 3.1: Judgements of Type System

We base the type system of the λ^H on the type system of the simple λ-calculus. The judgement:

$$\Gamma; \Delta \vdash_{\text{erase}} M : \tau$$

is an abbreviation of:

$$\text{erase}(\Gamma) \vdash \text{erase}(M) : \text{erase}(\tau)$$

in the type system of the simple λ-calculus.

3.4.1 Type Synthesis and Checking

We use a bidirectional approach to typing λ^H, inspired by [41].

We use judgements of form:

$$\Gamma; \Delta \vdash M^\star \Rightarrow \tau$$

to represent the synthesis of type τ from the given contexts Γ and Δ for term $M^\star$.

We use judgements of form:

$$\Gamma; \Delta \vdash M \Leftarrow \tau$$

to represent that the type τ typechecks under the given contexts Γ and Δ for term M.

In the former judgement, the type τ is an output of a type inference algorithm whereas in the latter judgement, the type τ is an input.

We show the main typing judgements for λ^H in Figure 3.4.

$$\frac{}{\Gamma, x : \{\nu : b \mid M\}, \Gamma'; \Delta \vdash x \Rightarrow \{\nu : b \mid \nu = x\}} \text{ TY-VAR-BASE}$$

$$\frac{}{\Gamma, x : (y : \tau_1) \to \tau_2, \Gamma'; \Delta \vdash x \Rightarrow (y : \tau_1) \to \tau_2} \text{ TY-VAR-FUNC}$$

$$\frac{}{\Gamma; \Delta \vdash c \Rightarrow \mathsf{Ty}(c)} \text{ TY-CONST}$$

$$\frac{\Gamma; \Delta \vdash M_1 \Rightarrow ((x : \tau_1) \to \tau_2) \quad \Gamma; \Delta \vdash M_2 \Leftarrow \tau_1}{\Gamma; \Delta \vdash M_1 M_2 \Rightarrow \tau_2[M_2/x]} \text{ TY-APP}$$

$$\frac{\Gamma; \Delta \vdash M \Leftarrow \tau}{\Gamma; \Delta \vdash (M : \tau) \Rightarrow \tau} \text{ TY-ANNO}$$

$$\frac{\Gamma, x : \tau_1; \Delta \vdash M \Leftarrow \tau_2 \quad \Gamma; \Delta \vdash ((x : \tau_1) \to \tau_2)}{\Gamma; \Delta \vdash \lambda x.M \Leftarrow ((x : \tau_1) \to \tau_2)} \text{ TY-ABS}$$

$$\frac{\Gamma; \Delta \vdash M_1 \Leftarrow \texttt{bool} \quad \Gamma; \Delta, M_1 \vdash M_2 \Leftarrow \tau \quad \Gamma; \Delta, \texttt{not}\ M_1 \vdash M_3 \Leftarrow \tau \quad \Gamma; \Delta \vdash \tau}{\Gamma; \Delta \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 \Leftarrow \tau} \text{ TY-COND}$$

$$\frac{\Gamma; \Delta \vdash \sigma <: \tau \quad \Gamma; \Delta \vdash M^\star \Rightarrow \sigma \quad \Gamma; \Delta \vdash \tau}{\Gamma; \Delta \vdash M^\star \Leftarrow \tau} \text{ TY-SUB}$$

Figure 3.4: Typing Judgements for λ^H

(TY-VAR-BASE) and (TY-VAR-FUNC) synthesise types for variables from the typing context. If the variable x has a base type in the typing context, a base type with refinement $\nu = x$ is synthesised for the variable. This allows the type to reference to the refinement attached to the variable in the context. If the variable x has a function type in the context, the function type is synthesised.

(TY-CONST) synthesises types for constant values. We list some example types for constants in Table 3.2. Integer and boolean literals have refinement types of their base type with predicate that the member is equal to the literal itself. Operators have refined function types, with return type refined with the predicate that the member is equal to the result of the operation.

(TY-APP) synthesises types for applications. Since the first term is in the syntactic category $M^\star$, it is always possible to synthesise a type for that term. The term must carry a function type $(x : \tau_1) \to \tau_2$. The second term needs to typecheck against τ_1, namely

the type of argument. The final synthesised type is $\tau_2[M_2/x]$ instead of τ_2. Without substitution τ_2 may contain variable x, which is no longer bound when the function type is eliminated, a substitution is thus necessary.

(TY-ANNO) synthesises types for annotated-terms. We use the typecheck judgement to verify the annotated type is correct, then return the annotated type as the result of synthesis. This rule makes it possible to synthesise types for checkable terms, as long as they are type-annotated.

1 ::	$\{\nu : \texttt{int} \mid \nu = 1\}$	integers
`true` ::	$\{\nu : \texttt{bool} \mid \nu = \texttt{true}\}$	booleans
$(+)$ ::	$(x : \texttt{int}) \to (y : \texttt{int}) \to \{\nu : \texttt{int} \mid \nu = x + y\}$	arithmetic
$(\geq)$ ::	$(x : \texttt{int}) \to (y : \texttt{int}) \to \{\nu : \texttt{bool} \mid \nu = (x \geq y)\}$	relational

Table 3.2: Constants and their Refined Types

For all typecheck judgements, it is required that the provided type is well-formed under the given context. We give well-formedness judgements for types in Figure 3.5, and explain them in Section 3.4.2.

(TY-ABS) checks types for abstractions. Here we assume without loss of generality that x does not occur in Γ. It is not possible to synthesise a type for abstractions without knowing the type of the argument, hence abstractions have a typecheck rule instead. When checked against a function type $(x : \tau_1) \to \tau_2$, the term M is typechecked against type τ_2, under the assumption that the abstraction variable x carries type τ_1 in the typing context. The result type τ_2 may contain the variable x, but the new context is able to provide type information for x.

(TY-COND) checks types for conditionals, i.e. boolean elimination. In this rule, we make use of the predicate context Δ. We model path sensitivity by adding path constraints into the predicate context Δ, which is encoded when deciding subtyping. The `then` branch carries predicate M_1, as it is taken when the predicate evaluates to `true`; whereas the `else` branch carries `not` M_1, as it is taken when the predicate evaluates to `false`.

(TY-SUB) checks types for synthesisable terms. The synthesised type for the term provides a single type for the term. Due to the presence of a subtyping relation, the term can carry more types than the unique synthesised type under the same typing context. The rule required that the synthesised type σ is a subtype of the specified type τ.

In a non-bidirectional typing setup, typing judgements usually contain a subsumption rule, which can be applied regardless of the syntax of the term. While the subsumption rule is powerful, the typing judgements are no longer syntax directed, and the subsumption rule complicates the implementation of the type system. In our setup, subtyping judgements are only used in (TY-SUB), when a synthesisable term is checked against a provided type.

3.4.2 Well-formedness of Types and Context

The well-formedness judgements for types are shown in Figure 3.5, of form:

$$\Gamma; \Delta \vdash \tau$$

The well-formedness judgements for contexts are shown in Figure 3.6, of form:

$$\vdash \Gamma; \Delta$$

In well-formedness judgements, we use the underlying type system of the simply typed λ-calculus instead of λ^H typing judgements to prevent cycles in inductive judgements.

$$\frac{\Gamma, \nu : b; \Delta \vdash_{\text{erase}} M : \texttt{bool} \quad \text{fv}(\{\nu : b \mid M\}) \subseteq \text{dom}(\Gamma)}{\Gamma; \Delta \vdash \{\nu : b \mid M\}} \text{ WF-TY-BASE}$$

$$\frac{\Gamma; \Delta \vdash \tau_1 \quad \Gamma, x : \tau_1; \Delta \vdash \tau_2 \quad \text{fv}((x : \tau_1) \to \tau_2) \subseteq \text{dom}(\Gamma)}{\Gamma; \Delta \vdash ((x : \tau_1) \to \tau_2)} \text{ WF-TY-FUNC}$$

Figure 3.5: Well-formedness Judgements for Types

For a type to be well-formed, the free variables in the type must be a subset of the domain of the typing context Γ. This ensures that type is closed under the typing context.

For refined base type (WF-TY-BASE), we use the type system for the simply typed λ-calculus to ensure that the predicate term M has a boolean type, under the assumption that the special variable for members ν carries the base type b.

For function type (WF-TY-FUNC), we check whether the argument type τ_1 is well-formed, as well as whether the result type τ_2 is well-formed with the assumption that x has type τ_1 in the context. This is necessary, since τ_2 may contain the variable x in the refinement predicate.

$$\frac{}{\vdash \varnothing; \varnothing} \text{ WF-CTX-BASE}$$

$$\frac{x \notin \text{dom}(\Gamma) \quad \vdash \Gamma; \Delta \quad \Gamma; \Delta \vdash \tau}{\vdash \Gamma, x : \tau; \Delta} \text{ WF-CTX-TYPE}$$

$$\frac{\vdash \Gamma; \Delta \quad \Gamma; \Delta \vdash_{\text{erase}} M : \texttt{bool}}{\vdash \Gamma; \Delta, M} \text{ WF-CTX-PRED}$$

Figure 3.6: Well-formedness Judgements of Context

For contexts to be well-formed, the variable context Γ should only contain well-formed types, and the predicate context Δ should contain well-typed boolean terms. We arrange the typing context in an ordered way such that when new variables are appended to the context, its type can refer to existing variables in the context. Similarly, typechecking in context well-formedness uses the type system under erasure.

(WF-CTX-BASE) provides the base case for well-formedness. Empty contexts are always well-formed.

(WF-CTX-TYPE) allows the typing context Γ to be extended. A variable with a well-formed type appended to a well-formed context preserves the well-formedness of the context. In our formalisation, all variables in the typing context are distinct.

(WF-CTX-PRED) allows the predicate context Δ to be extended. A boolean term M, typechecked under erasure in a well-formed context can be extended, with well-formedness preserved.

3.4.3 Subtyping and Encoding

Subtyping judgements are shown in Figure 3.7, of form:

$$\Gamma; \Delta \vdash \tau_1 <: \tau_2$$

We use a Satisfiability Modulo Theories (SMT) solver to decide the base cases in the subtyping judgements. We encode terms and typing contexts into a representation in logic, and then use the solver to check satisfiability of the encoded logical formula. While subtyping judgements are syntax-directed, the actual subtyping relation is semantics-driven, determined by the solver.

$$\frac{\mathsf{Valid}([\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])}{\Gamma, \Delta \vdash \{v : b \mid M_1\} <: \{v : b \mid M_2\}} \text{ SUB-BASE}$$

$$\frac{\Gamma; \Delta \vdash \sigma_1 <: \tau_1 \qquad \Gamma, x : \sigma_1; \Delta \vdash \tau_2 <: \sigma_2}{\Gamma, \Delta \vdash ((x : \tau_1) \to \tau_2) <: ((x : \sigma_1) \to \sigma_2)} \text{ SUB-FUNC}$$

Figure 3.7: Subtyping Judgements for λ^H

(SUB-BASE) defines the subtyping relation for base types. Two refined base type must first have the same base type, then we encode typing context Γ, predicate context Δ and refinements M_1, M_2 to form a logical formula:

$$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$$

We explain encoding later in this section. The formula is dispatched into a solver to check for validity. This gives rise to a semantically driven subtyping relation, where a type is a subtype of another, if the refinement for latter always holds whenever the refinement holds for the former [6].

(SUB-FUNC) defines subtyping relation for function types. We define subtyping for function types inductively, contravariant on the argument type and covariant on the result type.

We use an encoding for terms into SMT logic to decide subtyping relations. We define encoding of terms M, typing context Γ, and predicate context Δ in Definition 3.4.1.

Definition 3.4.1 (Encoding). *Encoding of terms* $[\![M]\!]$ *is defined as:*

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![c]\!] &= c_l \\
[\![\lambda x.M]\!] &= f_i \\
[\![M_1 M_2]\!] &= [\![M_1]\!]([\![M_2]\!]) \\
[\![M : \tau]\!] &= [\![M]\!] \\
[\![\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3]\!] &= ([\![M_1]\!] \implies [\![M_2]\!]) \wedge (\neg [\![M_1]\!] \implies [\![M_3]\!])
\end{aligned}
$$

Encoding of typing context $[\![\Gamma]\!]$ *is defined as:*

$$[\![\varnothing]\!] = \top \qquad [\![\Gamma, x : \{\nu : b \mid M\}]\!] = [\![\Gamma]\!] \wedge [\![M[x/\nu]]\!] \qquad [\![\Gamma, x : ((y : \tau_1) \to \tau_2)]\!] = [\![\Gamma]\!]$$

Encoding of predicate context $[\![\Delta]\!]$ *is defined as:*

$$[\![\varnothing]\!] = \top \qquad\qquad [\![\Delta, M]\!] = [\![\Delta]\!] \wedge [\![M]\!]$$

where c_l *is the correspondent constant in the logic,* f_i *is a fresh uninterpreted function.*

We encode the variables in λ^H into the variables in the SMT logic, as well as constants. We use a theory with booleans and integers, so that we can use boolean and integer constants in the logic, as well as the usual operators. Therefore, we encode all the constants in λ^H into corresponding literals or function in SMT.

For abstractions, we do not reflect the definition into the logic, instead we use uninterpreted function symbols without definition. This allows basic facts about function to be expressed in the logic, such as $x = y \implies fx = fy$ for a function f, but no extra information is available about the function f itself.

Function applications are encoded as function applications in the SMT logic. Type-annotated terms are encoded as if the term were not annotated. Conditional expressions are encoded as the conjunction of two implications, representing the two possible branches.

For typing contexts, we encode refined base types in Γ into a formula involving the refinement attached in the type. The special variable ν, representing members of the type, is substituted to the variable with that type in the typing context Γ. This allows the refinement to be correctly attached the variable at encoding. Variables carrying a function type in the typing contexts are ignored in this current setup.

Since the predicate context Δ consists of zero or more terms, we simply encode the terms in the predicate context one by one.

We show that the subtyping relation is reflexive and transitive.

Theorem 3.4.2. *Subtyping is reflexive.* $\Gamma; \Delta \vdash \tau <: \tau$

Proof. By induction on structure of τ.

1. $\tau = \{\nu : b \mid M\}$

   It is straightforward to show that $[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M]\!] \implies [\![M]\!]$ is valid. By (SUB-BASE), we have $\Gamma; \Delta \vdash \{\nu : b \mid M\} <: \{\nu : b \mid M\}$, as required.

2. $\tau = (x : \tau_1) \to \tau_2$

   By inductive hypothesis.

□

Theorem 3.4.3. *Subtyping is transitive. If* $\Gamma; \Delta \vdash \tau <: \sigma$ *and* $\Gamma; \Delta \vdash \sigma <: \rho$*, then* $\Gamma; \Delta \vdash \tau <: \rho$

Proof. We notice first that there is no derivation for $\Gamma; \Delta \vdash \{\nu : b \mid M\} <: (x : \tau_1) \to \tau_2$, nor $\Gamma; \Delta \vdash (x : \tau_1) \to \tau_2 <: \{\nu : b \mid M\}$. Hence τ, σ and ρ are either all refined base types, or all function types.

1. $\tau = \{\nu : b \mid M_1\}, \sigma = \{\nu : b \mid M_2\}, \rho = \{\nu : b \mid M_3\}$

   From (SUB-BASE), we know that $[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$, and that $[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_2]\!] \implies [\![M_3]\!]$ are both valid.

   We show that $[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_3]\!]$ is valid.

1.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$	premise
2.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_2]\!] \implies [\![M_3]\!]$	premise
3.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!]$	assumption
4.	$[\![M_2]\!]$	$\implies E(1,3)$
5.	$[\![\Gamma]\!] \wedge [\![\Delta]\!]$	$\wedge E_1(3)$
6.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_2]\!]$	$\wedge I(5,4)$
7.	$[\![M_3]\!]$	$\implies E(2,6)$
8.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_3]\!]$	$\implies I(3,8)$

   By (SUB-BASE), we have $\Gamma; \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_3\}$, as required.

2. $\tau = (x : \tau_1) \to \tau_2, \sigma = (x : \sigma_1) \to \sigma_2, \rho = (x : \rho_1) \to \rho_2$

   By inductive hypothesis.

□

3.4.4 Typing under Erasure

Well-typed terms in λ^H are well-typed in the simply typed λ-calculus under erasure, as presented in Theorem 3.4.4. Practically, this means an implementation can typecheck terms in λ^H, and then erase all the refinements and use the underlying type system of the simply typed λ-calculus, so that no runtime information about refinements need to be kept.

Theorem 3.4.4 (Typeable under Erasure)**.** *If* $\Gamma; \Delta \vdash M \Rightarrow \tau$ *or* $\Gamma; \Delta \vdash M \Leftarrow \tau$*, then* $\Gamma; \Delta \vdash_{erase} M : \tau$.

Proof. By mutual induction on the derivation of $\Gamma; \Delta \vdash M \Rightarrow \tau$ and $\Gamma; \Delta \vdash M \Leftarrow \tau$.

1. $\Gamma; \Delta \vdash x \Rightarrow \{\nu : b \mid M\}$ (TY-VAR-BASE)

   From (TY-VAR-BASE), we have $\Gamma = \Gamma_1, x : \{\nu : b \mid M'\}, \Gamma_2$ for some Γ_1, Γ_2, M'.

   By definition, we have $\text{erase}(\Gamma) = \text{erase}(\Gamma_1), x : b, \text{erase}(\Gamma_2)$; and $\text{erase}(\{\nu : b \mid M\}) = b$, as required.

   Hence, we have $\text{erase}(\Gamma_1), x : b, \text{erase}(\Gamma_2) \vdash x : b$.

2. $\Gamma; \Delta \vdash x \Rightarrow (y : \tau_1) \to \tau_2$ (TY-VAR-FUNC)

   From (TY-VAR-BASE), we have $\Gamma = \Gamma_1, x : (y : \tau_1 \to \tau_2), \Gamma_2$ for some Γ_1, Γ_2, τ_1, τ_2.

   By definition , we have $\text{erase}(\Gamma) = \text{erase}(\Gamma_1), x : (\text{erase}(\tau_1) \to \text{erase}(\tau_2)), \text{erase}(\Gamma_2)$; and $\text{erase}((y : \tau_1) \to \tau_2) = \text{erase}(\tau_1) \to \text{erase}(\tau_2)$.

   Hence, we have $\text{erase}(\Gamma_1), x : (\text{erase}(\tau_1) \to \text{erase}(\tau_2)), \text{erase}(\Gamma_2) \vdash x : (\text{erase}(\tau_1) \to \text{erase}(\tau_2))$, as required.

3. $\Gamma; \Delta \vdash c \Rightarrow \mathsf{Ty}(c)$ (TY-CONST)

   The constants are typeable under erasure with their erased types.

4. $\Gamma; \Delta \vdash M_1 M_2 \Rightarrow \tau$ (TY-APP)

   From (TY-APP), we have $\Gamma; \Delta \vdash M_1 \Rightarrow (x : \tau_1) \to \tau_2$ and $\Gamma; \Delta \vdash M_2 \Leftarrow \tau_1$.

   By inductive hypothesis, we have $\Gamma; \Delta \vdash_{\text{erase}} M_1 : (x : \tau_1) \to \tau_2$, i.e. $\text{erase}(\Gamma) \vdash \text{erase}(M_1) : \text{erase}(\tau_1) \to \text{erase}(\tau_2)$; and that $\Gamma; \Delta \vdash_{\text{erase}} M_2 : \tau_1$, i.e. $\text{erase}(\Gamma) \vdash \text{erase}(M_2) : \text{erase}(\tau_1)$.

   Hence we have $\text{erase}(\Gamma) \vdash \text{erase}(M_1)\,\text{erase}(M_2) : \text{erase}(\tau_2)$, i.e. $\text{erase}(\Gamma) \vdash \text{erase}(M_1 M_2) : \text{erase}(\tau_2)$ as required.

5. $\Gamma; \Delta \vdash (M : \tau) \Rightarrow \tau$ (TY-ANNO)

   By (TY-ANNO), we have $\Gamma; \Delta \vdash M \Leftarrow \tau$.

   By inductive hypothesis, we have $\Gamma; \Delta \vdash_{\text{erase}} M : \tau$.

   Since $\text{erase}(M : \tau) = \text{erase}(M)$, we have $\Gamma; \Delta \vdash_{\text{erase}} (M : \tau) : \tau$.

6. $\Gamma; \Delta \vdash (\lambda x.M) \Leftarrow (x : \tau_1) \to \tau_2$ (TY-ABS)

   By (TY-ABS), we have $\Gamma, x : \tau_1; \Delta \vdash M \Leftarrow \tau_2$.

   By inductive hypothesis, we have $\Gamma, x : \tau_1 \vdash_{\text{erase}} M : \tau_2$, i.e. $\text{erase}(\Gamma), x : \text{erase}(\tau_1) \vdash \text{erase}(M) : \text{erase}(\tau_2)$.

   We have $\text{erase}(\Gamma) \vdash \lambda x.\,\text{erase}(M) : \text{erase}(\tau_1) \to \text{erase}(\tau_2)$. Since $\text{erase}(x : \tau_1 \to \tau_2) = \text{erase}(\tau_1) \to \text{erase}(\tau_2)$, and $\text{erase}(\lambda x.M) = \lambda x.\,\text{erase}(M)$, we have $\Gamma; \Delta \vdash_{\text{erase}} \lambda x.M \Leftarrow (x : \tau_1) \to \tau_2$.

7. $\Gamma; \Delta \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 \Leftarrow \tau$ (TY-COND)

   By (TY-COND), we have $\Gamma; \Delta \vdash M_1 \Leftarrow \texttt{bool}$ and $\Gamma; \Delta, M_1 \vdash M_2 \Leftarrow \tau$ and $\Gamma; \Delta, \texttt{not}\ M_2 \vdash M_3 \Leftarrow \tau$.

By inductive hypothesis, we have $\Gamma; \Delta \vdash_{\text{erase}} M_1 : \texttt{bool}$, i.e. $\text{erase}(\Gamma) \vdash \text{erase}(M_1) : \texttt{bool}$.

Similarly, We have $\Gamma; \Delta, M_2 \vdash_{\text{erase}} M_2 : \tau$, i.e. $\text{erase}(\Gamma) \vdash \text{erase}(M_2) : \text{erase}(\tau)$; and $\Gamma; \Delta, M_2 \vdash_{\text{erase}} M_2 : \tau$, i.e. $\text{erase}(\Gamma) \vdash \text{erase}(M_2) : \text{erase}(\tau)$. We notice typing under erasure does not depend on the predicate context Δ.

Hence we have $\text{erase}(\Gamma) \vdash \texttt{if}\ \text{erase}(M_1)\ \texttt{then}\ \text{erase}(M_2)\ \texttt{else}\ \text{erase}(M_3) : \text{erase}(\tau)$, i.e. $\Gamma; \Delta \vdash_{\text{erase}} \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 \Leftarrow \tau$.

8. $\Gamma; \Delta \vdash M \Leftarrow \tau$ (TY-SUB)

   By (TY-SUB), we have $\Gamma; \Delta \vdash M \Rightarrow \sigma$ and $\Gamma; \Delta \vdash \sigma <: \tau$.

   By inductive hypothesis, we have $\Gamma; \Delta \vdash_{\text{erase}} M : \sigma$.

   By Lemma B.1.1, we have $\text{erase}(\tau) = \text{erase}(\sigma)$.

   Hence we have $\Gamma; \Delta \vdash_{\text{erase}} M : \tau$, as required.

□

3.5 Type Safety

We prove the type safety theorems of λ^H in this section. Type safety is best known by the slogan "Well-typed programs cannot go wrong" [32]. We present the main theorems, preservation in Theorem 3.5.3 and progress in Theorem 3.5.5. We include lemmas and their proofs in Appendix B.1. Preservation theorem states that type for terms are preserved under reduction. Progress theorem states that well-typed terms are either values or can further reduce. Combined together, we achieve type safety, that if a term is well-typed, it cannot reach a stuck state where the term is not a value and cannot reduce.

Lemma 3.5.1 (Substitution (Synthesis))**.** *Suppose* $\Gamma, \Gamma'; \Delta \vdash M \Rightarrow \tau$.

1. *If* $\Gamma, x : \tau, \Gamma'; \Delta \vdash N \Rightarrow \sigma$, *then* $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash N[M/x] \Rightarrow \sigma[M/x]$
2. *If* $\Gamma, x : \tau, \Gamma'; \Delta \vdash N \Leftarrow \sigma$, *then* $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash N[M/x] \Leftarrow \sigma[M/x]$

Proof. By simultaneous induction on typing derivations $\Gamma, x : \tau, \Gamma'; \Delta \vdash N \Rightarrow \sigma$ and $\Gamma, x : \tau, \Gamma'; \Delta \vdash N \Leftarrow \sigma$.

1. $\Gamma, x : \tau, \Gamma'; \Delta \vdash y \Rightarrow \sigma$ (TY-VAR-BASE), (TY-VAR-FUNC)

   When $x = y$, $y[M/x] = M$. From (TY-VAR), it must have been the case that $\tau = \sigma$. We then show that $\Gamma, \Gamma'[M/x] \vdash M \Rightarrow \sigma[M/x]$ in Lemma B.1.12.

   When $x \neq y$, $y[M/x] = y$. We show that $\Gamma, \Gamma'[M/x] \vdash y \Rightarrow \sigma[M/x]$ in Lemma B.1.8.

2. $\Gamma, x : \tau, \Gamma'; \Delta \vdash c \Rightarrow \sigma$ (TY-CONST)

   We assume that the typing for constants does not depend on the typing context, and the types for constants contains no free variables for substitution. Hence we are left to show $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash c \Rightarrow \sigma$, which is easy by (TY-CONST).

3. $\Gamma, x : \tau, \Gamma'; \Delta \vdash \lambda y.N' \Leftarrow ((y : \sigma_1) \to \sigma_2)$ (TY-ABS)

   From (TY-ABS) we have $\Gamma, x : \tau, \Gamma', y : \sigma_1; \Delta \vdash N' \Leftarrow \sigma_2$ and $\Gamma, x : \tau, \Gamma'; \Delta \vdash ((y : \sigma_1) \to \sigma_2)$.

   From inductive hypothesis, we have $\Gamma, \Gamma'[M/x], y : \sigma_1[M/x]; \Delta[M/x] \vdash N'[M/x] \Leftarrow \sigma_2[M/x]$.

   By Lemma B.1.9, we have $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash ((y : \sigma_1) \to \sigma_2)[M/x]$. Note that $((y : \sigma_1) \to \sigma_2)[M/x] = ((y : \sigma_1[M/x]) \to \sigma_2[M/x])$.

   We apply (TY-ABS) and obtain $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash N'[M/x] \Leftarrow ((y : \sigma_1) \to \sigma_2)[M/x]$, as required.

4. $\Gamma, x : \tau, \Gamma'; \Delta \vdash N_1 N_2 \Rightarrow \sigma$ (TY-APP)

   From (TY-APP) we have $\Gamma, x : \tau, \Gamma'; \Delta \vdash N_1 \Rightarrow ((y : \sigma_1) \to \sigma_2)$ and $\Gamma, x : \tau, \Gamma'; \Delta \vdash N_2 \Leftarrow \sigma_1$ for some σ_1 and σ_2 with $\sigma_2[M_2/x] = \sigma$. Note that here y is fresh.

   From inductive hypothesis, we have $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash N_1[M/x] \Rightarrow ((y : \sigma_1) \to \sigma_2)[M/x]$ and $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash N_2[M/x] \Leftarrow \sigma_1[M/x]$.

   Note that $((y : \sigma_1) \to \sigma_2)[M/x] = ((y : \sigma_1[M/x]) \to \sigma_2[M/x])$, hence we can apply both rules to (TY-APP) and obtain $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash N_1 N_2 \Rightarrow \sigma_2[M/x]$. Since $\sigma_2[M/x] = \sigma$, we have shown the goal.

5. $\Gamma, x : \tau, \Gamma'; \Delta \vdash \texttt{if } N_1 \texttt{ then } N_2 \texttt{ else } N_3 \Leftarrow \sigma$ (TY-COND)

   By inductive hypothesis and Lemma B.1.9.

6. $\Gamma, x : \tau, \Gamma'; \Delta \vdash (N' : \sigma) \Rightarrow \sigma$ (TY-ANNO)

   By inductive hypothesis.

7. $\Gamma, x : \tau, \Gamma; \Delta \vdash N \Leftarrow \sigma$ (TY-SUB)

   By inductive hypothesis, Lemma B.1.9 and Lemma B.1.11.

□

Corollary 3.5.2 (Substitution (Typechecking))**.** *Suppose* $\Gamma, \Gamma'; \Delta \vdash M \Leftarrow \tau$.

1. *If* $\Gamma, x : \tau, \Gamma'; \Delta \vdash N \Rightarrow \sigma$, *then* $\Gamma, \Gamma'[M : \tau/x]; \Delta[M : \tau/x] \vdash N[M : \tau/x] \Rightarrow \sigma[M : \tau/x]$

2. *If* $\Gamma, x : \tau, \Gamma'; \Delta \vdash N \Leftarrow \sigma$, *then* $\Gamma, \Gamma'[M : \tau/x]; \Delta[M : \tau/x] \vdash N[M : \tau/x] \Leftarrow \sigma[M : \tau/x]$

Proof. By (TY-ANNO), we have $\Gamma, \Gamma'; \Delta \vdash (M : \tau) \Rightarrow \tau$, the results follow from Lemma 3.5.1. □

Theorem 3.5.3 (Preservation)**.** *Suppose* $M \to M'$.

1. *If* $\Gamma; \Delta \vdash M \Rightarrow \tau$, *then* $\Gamma; \Delta \vdash M' \Rightarrow \tau$
2. *If* $\Gamma; \Delta \vdash M \Leftarrow \tau$, *then* $\Gamma; \Delta \vdash M' \Leftarrow \tau$

Proof. By induction on the reduction relation $M \to M'$.

1. $(\lambda x.M_1 : ((x : \tau_1) \to \tau_2))v_2 \to M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]$ (RED-APP-ANNO)

   Without loss of generality, we let x be fresh in $\Gamma; \Delta$.

   - Suppose $\Gamma; \Delta \vdash (\lambda x.M_1 : ((x : \tau_1) \to \tau_2))v_2 \Rightarrow \tau$, we show that $\Gamma; \Delta \vdash (M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]) \Rightarrow \tau$.

     From the rule (TY-APP), we have $\Gamma; \Delta \vdash (\lambda x.M_1 : ((x : \tau_1) \to \tau_2)) \Rightarrow ((x : \tau_1) \to \tau_2)$, and $\Gamma, x : \tau_1; \Delta \vdash v_2 \Leftarrow \tau_1$ and $\tau = \tau_2[v_2 : \tau_1/x]$.

     From the rule (TY-ANNO), we have $\Gamma; \Delta \vdash \lambda x.M_1 \Leftarrow ((x : \tau_1) \to \tau_2)$.

     From the rule (TY-ABS), we have $\Gamma, x : \tau_1; \Delta \vdash M_1 \Leftarrow \tau_2$.

     By Corollary 3.5.2, we have $\Gamma; \Delta[v_2 : \tau_1/x] \vdash M_1[v_2 : \tau_1/x] \Leftarrow \tau_2[v_2 : \tau_1/x]$. Since x is fresh in $\Gamma; \Delta$, we have $\Delta[v_2 : \tau_1/x] = \Delta$.

     By (TY-ANNO), we have $\Gamma; \Delta \vdash (M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]) \Rightarrow \tau_2[v_2 : \tau_1/x]$. Note that $\tau = \tau_2[v_2 : \tau_1/x]$, so we have $\Gamma; \Delta \vdash M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x] \Rightarrow \tau$, as required.
   - Suppose $\Gamma; \Delta \vdash (\lambda x.M_1 : ((x : \tau_1) \to \tau_2))v_2 \Leftarrow \tau$, we show that $\Gamma; \Delta \vdash (M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]) \Leftarrow \tau$.

     From (TY-SUB), we have $\Gamma; \Delta \vdash (\lambda x.M_1 : ((x : \tau_1) \to \tau_2)) \Rightarrow \sigma$, and $\Gamma; \Delta \vdash \tau$ and $\Gamma; \Delta \vdash \sigma <: \tau$, for some type σ.

     By applying the result from the previous part, we have $\Gamma; \Delta \vdash (M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]) \Rightarrow \sigma$.

     By (TY-SUB), we have $\Gamma; \Delta \vdash (M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]) \Leftarrow \tau$, as required.

2. `if true then` M_2 `else` $M_3 \to M_2$ (RED-IF-TRUE)

   - There is no derivation for $\Gamma; \Delta \vdash$ `if true then` M_2 `else` $M_3 \Rightarrow \tau$.
   - Suppose $\Gamma; \Delta \vdash$ `if true then` M_2 `else` $M_3 \Leftarrow \tau$, we show that $\Gamma; \Delta \vdash M_2 \Leftarrow \tau$.

     From (TY-COND), we have $\Gamma; \Delta, \texttt{true} \vdash M_2 \Leftarrow \tau$. We know that $[\![\Delta, \texttt{true}]\!] = [\![\Delta]\!] \wedge \top$, hence we have $[\![\Delta]\!] \implies [\![\Delta, \texttt{true}]\!]$. By Lemma B.1.16, we have $\Gamma; \Delta \vdash M_2 \Leftarrow \tau$.

3. `if false then` M_2 `else` $M_3 \to M_3$ (RED-IF-FALSE)

   Similar to the case of (RED-IF-TRUE), except that we have `not false` for the truthy value instead of `true`.

4. If $M \to M'$, then $C[M] \to C[M']$ (RED-CTX)

   By inductive hypothesis.

□

Corollary 3.5.4 (Preservation (Multistep)). *Suppose $M \to^* M'$.*

1. *If $\Gamma; \Delta \vdash M \Rightarrow \tau$, then $\Gamma; \Delta \vdash M' \Rightarrow \tau$*
2. *If $\Gamma; \Delta \vdash M \Leftarrow \tau$, then $\Gamma; \Delta \vdash M' \Leftarrow \tau$*

Proof. If $M = M'$ (reflexive case), the conclusion holds trivially.

If $M \to M'$ (single step case), the conclusion holds by Theorem 3.5.3.

If $M \to M''$ and $M'' \to^* M'$ (transitive case), the conlusion holds by Theorem 3.5.3 and inductive hypothesis. □

Theorem 3.5.5 (Progress). *If $\varnothing; \varnothing \vdash M \Rightarrow \tau$ or $\varnothing; \varnothing \vdash M \Leftarrow \tau$, then either M is a value, or there exists M' such that $M \to M'$.*

Proof. By mutual induction on the derivation of $\varnothing; \varnothing \vdash M \Rightarrow \tau$ and $\varnothing; \varnothing \vdash M \Leftarrow \tau$.

1. (TY-VAR-BASE), (TY-VAR-FUNC)

   There is no derivation of such rules, since typing context Γ is empty.

2. $\varnothing; \varnothing \vdash c \Rightarrow \tau$ (TY-CONST)

   c is a value.

3. $\varnothing; \varnothing \vdash M_1 M_2 \Rightarrow \tau$ (TY-APP)

   By (TY-APP), we have $\varnothing; \varnothing \vdash M_1 \Rightarrow (x : \tau_1) \to \tau_2$ and $\varnothing; \varnothing \vdash M_2 \Leftarrow \tau_1$ for some τ_1 and τ_2 such that $\tau_2[M_1/x] = \tau$.

   By inductive hypothesis, we have that M_1 is either a value v_1 or there exists M_1' such that $M_1 \to M_1'$.

   - By inductive hypothesis, we have that M_2 is either a value v_2 or there exists M_2' such that $M_2 \to M_2'$.
     - We have value v_1 such that $\varnothing; \varnothing \vdash v_1 \Rightarrow (x : \tau_1) \to \tau_2$. By inversion, we know that $v_1 = \lambda x.M : (x : \tau_1) \to \tau_2$ for some term M. By (RED-APP-ANNO), we have $(\lambda x.M_1 : ((x : \tau_1) \to \tau_2))v_2 \to M_1[v_2 : \tau_1/x] : \tau_2[v_2 : \tau_1/x]$.
     - By (RED-CTX), we have $v_1 M_2 \to v_1 M_2'$.
   - By (RED-CTX), we have $M_1 M_2 \to M_1' M_2$.

4. $\varnothing; \varnothing \vdash (M : \tau) \Rightarrow \tau$ (TY-ANNO)

   From (TY-ANNO), we have $\varnothing; \varnothing \vdash M \Leftarrow \tau$.

   By inductive hypothesis, we have that either M is a value v or there exists M' such that $M \to M'$.

   - $v : \tau$ is a value.
   - By (RED-CTX), we have $(M : \tau) \to (M' : \tau)$.

5. $\varnothing; \varnothing \vdash \lambda x.M \Leftarrow \tau$ (TY-ABS)

   $\lambda x.M$ is a value.

6. $\varnothing; \varnothing \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 \Leftarrow \tau$ (TY-COND)

   From (TY-COND), we have $\varnothing; \varnothing \vdash M_1 \Leftarrow \texttt{bool}$.

   By inductive hypothesis, we have M_1 is either a value v_1, or there exists M_1' such that $M_1 \to M_1'$.

   - By inversion, we have $v_1 = \texttt{true}$ or $v_1 = \texttt{false}$.
     - $v_1 = \texttt{true}$
       By (RED-IF-TRUE), we have $\texttt{if true then}\ M_2\ \texttt{else}\ M_3 \to M_2$.
     - $v_1 = \texttt{false}$
       By (RED-IF-FALSE), we have $\texttt{if false then}\ M_2\ \texttt{else}\ M_3 \to M_3$.
   - By (RED-CTX),
     we have $\texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 \to \texttt{if}\ M_1'\ \texttt{then}\ M_2\ \texttt{else}\ M_3$.

7. $\varnothing; \varnothing \vdash M \Leftarrow \tau$ (TY-SUB)

   By (TY-SUB), we have $\varnothing; \varnothing \vdash M \Rightarrow \sigma$ for some σ.

   By inductive hypothesis, M is either a value v or there exists M' such that $M \to M'$.

□

Based on the type safety results, we finally prove a theorem relating well-typed terms and their refinement types. In this theorem, we show that a term carrying a refinement type respects the predicate in the refinement, if the term reduces to a value. This makes the connection between the terms and their refinement types, showing the validity of predicates in our type system.

Theorem 3.5.6 (Predicate Validity)**.** *If* $\varnothing; \varnothing \vdash M \Leftarrow \{\nu : b \mid N\}$ *and* $M \to^* v$*, then* $[\![N]\!][v/\nu]$ *is valid.*

Proof. By Corollary 3.5.4, we have $\varnothing; \varnothing \vdash v \Leftarrow \{\nu : b \mid N\}$.

By inversion of values, we have that v is a constant literal of base type b. Therefore, by (TY-CONST), we have $\varnothing; \varnothing \vdash v \Rightarrow \{\nu : b \mid \nu = v\}$.

From (TY-SUB), we have that $\varnothing; \varnothing \vdash \{\nu : b \mid \nu = v\} <: \{\nu : b \mid N\}$.

From (SUB-BASE), we have that the following formula is valid:

$$[\![\varnothing]\!] \wedge [\![\varnothing]\!] \wedge [\![v = v]\!] \implies [\![N]\!]$$

Note that $[\![\varnothing]\!] = \top$, we simplify the formula to:

$$[\![v = v]\!] \implies [\![N]\!]$$

By substitution, we have $[\![N]\!][v/v]$ is valid. □

Chapter 4

FLUIDTYPES: A Type System Implementation of λ^H

In this chapter, we present FLUIDTYPES, an F# library that extends the F# type system with refinement types. In refinement types, we allow base types to be refined by boolean predicates. Refinement types can be annotated explicitly in F# source codes and can be checked by the typechecker extension we develop. We explain the implementation details and justify design choices in this chapter.

FLUIDTYPES works stand alone, allowing programmers to use the library for refinement types in their code; but in combination with our code generator for protocols with refinements, we can verify their implementation by static typechecking. We discuss the details in Chapter 5.

This part of the project is open source and the source code is publicly available at `https://github.com/fangyi-zhou/FluidTypes` under the MIT license.

The library is structured as three modules, corresponding to three major functionalities:

- `Refinements`: A typechecker for λ^H
- `Extractions`: For the extraction of F# programs into terms in λ^H
- `Annotations`: For ways to annotations to express refinements in source code

In order to support a wide range of language features of F#, the core calculus of λ^H is extended with a few practical extension, including type aliases, record types, and enumerations.

We explain implementation details of `Refinements` in Section 4.1, `Extractions` in Section 4.2, `Annotations` in Section 4.3, and type system extensions in Section 4.4.

4.1 Implementation of λ^H Type System

In Section 3.4, we formalised the type system for λ^H. he typing judgements are presented in a bidirectional style, which has the benefit of being syntactic-directed. The

implementation of type system can hence be constructed in a straightforward fashion.

4.1.1 Data Type Definitions

We use mutually recursive algebraic data types to represent the terms M and types τ in λ^H, since a term can contain a type (for type-annotated terms $(M : \tau)$) and a type can contain a term (for refined base types $\{\nu : b \mid M\}$).

For typing contexts Γ, we use a mapping from variable to types. In the definition, entries in the typing context are ordered. From the well-formedness judgements of contexts, a typing context is only well-formed when there exists an sequence of variables, such that for any variable in the sequence, the type for that variable in the context may only contain free variables occurring earlier in the sequence, and must not contain free variables occuring later in that sequence. This means a typing context such as $x : \{\nu : \texttt{int} \mid \nu = y\}, y : \{\nu : \texttt{int} \mid \nu = x\}, \varnothing$ is not well-formed.

Practically, the ordering in the typing context can be ignored safely. The well-formedness property of the typing context is only affected when a new entry is added to the typing context. When an entry is added, no free variable other than those already in the typing context can occur in the type of that entry. Once this property is checked when new entry is added, the well-formedness of the typing context is preserved. On the other hand, we always start with an empty context at the beginning, and an empty context is always well-formed. Instead of using a list data structure and keeping the ordering of variables in the typing context, we can ignore the ordering by using a map data structure.

4.1.2 Main Typing Judgements

The main typing judgements are the type synthesis judgement $(\Gamma; \Delta \vdash M \Rightarrow \tau)$ and type checking $(\Gamma; \Delta \vdash M \Leftarrow \tau)$ judgement, in Figure 3.4. They are defined as mutually recursive functions with the signatures as shown in Figure 4.1.

```
1 val infer_type: Ctx → Term → Ty option
2 val check_type: Ctx → Term → Ty → bool
```

Figure 4.1: Function Signatures of Main Typing Judgements

In the type signature, `Ctx` stands for the typing context and the predicate context $(\Gamma; \Delta)$, `Term` is the type for term M, and `Ty` is the type for type τ. The type inference function returns an option type, where a value present corresponds to the cases where a type can be synthesised, and absent otherwise. In cases when a type cannot be inferred, we report an error to the user. The type check function returns a boolean value indicating whether the given term admits the given type.

Algorithm 1 shows the implementation of main typing judgements. Details on how other judgements and definitions are implemented are not shown, for instance,

```
let rec infer_type (Γ; Δ) M =
    match M with
    | x when x ∈ Γ →
        match Γ(x) with
        | {ν : b | _} → Some {ν : b | ν = x}
        | (x : τ1) → τ2 → Some (x : τ1) → τ2
    | x → None
    | c → Some Ty(c)
    | M1 M2 →
        match infer_type (Γ; Δ) M1 with
        | Some ((x : τ1) → τ2) →
            if check_type (Γ; Δ) M2 τ1 then Some τ2[M2/x]
            else None
        | None → None
    | (M : τ) →
        if check_type (Γ; Δ) M τ then Some τ
        else None
    | _ → None
and check_type (Γ; Δ) M τ =
    if is_type_wellformed (Γ; Δ) τ then
        match M with
        | λx.M →
            match τ with
            | (x : τ1) → τ2 → check_type (Γ, x : τ1; Δ) M τ2
            | _ → false
        | if M1 then M2 else M3 →
            check_type (Γ; Δ) M1 bool
            && check_type (Γ; Δ, M1) M2 τ
            && check_type (Γ; Δ, not M1) M3 τ
        | M →
            match infer_type (Γ; Δ) M with
            | Some σ → is_subtype (Γ; Δ) σ τ
            | None → false
    else false
```

Algorithm 1: Type inference and checking algorithm

substitution of terms and well-formedness of types, since these definitions or judgements can be similarly implemented in a straightforward way. In the algorithm shown, *is_type_wellformed* $(\Gamma; \Delta)\ \tau$ stands for the type well-formedness judgement $\Gamma; \Delta \vdash \tau$ and *is_subtype* $(\Gamma; \Delta)\ \tau_1\ \tau_2$ stands for the subtyping judgement $\Gamma; \Delta \vdash \tau_1 <: \tau_2$.

The benefit of bidirectional typing judgements is evident, as they easily convert to algorithms. Since the recursive invocation of the typing functions are on structurally smaller terms M, the algorithm terminates, provided that the terms are finite themselves. (F# is strict and we do not define infinite data structures).

We do not go into implementation details of well-formness judgements of types and typing judgements under erasure here, since they can be converted into algorithms in similar ways. Curious readers may refer to the implementation file `src/Refinements/Typing.fs` for more details.

4.1.3 Subtyping Judgements and SMT Encodings

Another core component of the type system is the subtyping judgement. This is decided by an external solver. We produce standarised SMT-LIB scripts [2] for SMT queries to decide subtyping. This allows us to use different SMT solvers supporting the standard. In this implementation, we use the SMT solver Z3 [10].

We recall that the base case of the subtyping judgement is as follows:

$$\frac{\mathsf{Valid}(\llbracket\Gamma\rrbracket \wedge \llbracket\Delta\rrbracket \wedge \llbracket M_1\rrbracket \implies \llbracket M_2\rrbracket)}{\Gamma, \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}}\ \text{SUB-BASE}$$

SMT solvers are able to check whether a conjunction of formulas are satisfiable. We formalised the subtyping judgement with logical formulas in validity form. So we need to use the equivalent *satisfiability* form of the formula. That is to say, the formula for the SMT solver to decide is:

$$\llbracket\Gamma\rrbracket \wedge \llbracket\Delta\rrbracket \wedge \llbracket M_1\rrbracket \wedge \neg\llbracket M_2\rrbracket \tag{†}$$

We produce a SMT-LIB script for each subtyping check. This script consists of variable declarations, assertions and a command according to the standard. The script is dispatched to Z3, but the end user is able to use their favourite solver, as long as it support the standard.

For variable declarations, we define variables and their sorts in our logical formula. We map λ^H base types into appropriate logical sorts. We use a theory with integers and booleans in our setup, and declare `Int` and `Bool` sorts for variables carrying integer and boolean base types in λ^H respectively. Using `declare-const` command, we declare variables involved in the formula, i.e. variables in the current typing context Γ and the special variable ν representing the value of the refinement type.

For assertions, we encode Equation (†) into SMT assertions using `assert` command. Since there is no encoding for function values and function applications other than those

defined as constants, we call these terms *unencodable* and we use a conservative approach to approximate them, ensuring soundness at the cost of completeness.

In the end of the script, we use `check-sat` command to instruct the solver to check for satisfiability of the formula. If the solver decides that the formula is not satisfiable (i.e. `UNSAT`), then the subtyping judgement holds. Otherwise, we can query the SMT solver for an example of the assignment to the variables, and use the example for error messages later.

When an unencodable term appears in Γ, Δ or M_1, we ignore the unencodable term. If the formula without the unencodable term were unsatisfiable, then the formula with that ignored term encoded must also be unsatisfiable, since the latter formula is weaker than the former. When an unencodable term appears in M_2, we directly come to the conclusion that subtyping judgement does not hold. The approach leads to incompleteness, since we are not able some subtype judgements when they hold due to approximation, e.g.

$$f : (x : \texttt{int}) \to \texttt{int}, z : \texttt{int}; \varnothing \vdash \{\nu : \texttt{int} \mid f\ z = 0\} <: \{\nu : \texttt{int} \mid f\ z \geq 0\}$$

4.1.4 Summary

We described the `Refinements` module of FLUIDTYPES, which implements typing judgements of λ^H. We use a sound approximation of the encoding of λ^H terms into SMT formulas for deciding subtyping judgements.

For future work, it is worth investigating how to encode abstractions and function applications. In Definition 3.4.1, we use uninterpreted function symbols to encode abstractions. Practically, SMT-LIB has support for functions. We can declare functions in SMT-LIB with different arities with builtin sorts for integers and booleans. However, application of functions cannot be partial in SMT-LIB.

In addition, it is interesting to investigate how function types in the typing context can be encoded. Despite definition being absent, a function type can be encoded as a first logic formula with universal quantifiers. For example, a function f with type $(x : \{\nu : \texttt{int} \mid \nu > 0\}) \to \{\nu : \texttt{int} \mid \nu \leq x\}$ can produce the term $\forall x : \texttt{Int}.(x > 0) \implies (f\,x \leq x)$, which establishes the pre- and post-conditions of the function.

In SMT-LIB, integers are unbounded mathematical integers. In F#, integers are bounded and has wrap around behaviour when overflowing. The discrepancy of integer behaviour in SMT-LIB and F# can be addressed by addressed by using bit vectors instead of integers in SMT-LIB to represent integers.

In our current implementation, we do not optimise queries to the SMT solver. We may be able to handle small programs in our examples, but this may impose scalability issues for large use cases. As future work, we can deploy optimisation techniques on SMT queries, for example those used by other verification tools [8].

4.2 From F# Expression to λ^H Terms

In order to use the type system for λ^H, we first convert the F# code to λ^H terms, and then use the typechecker for λ^H for validation. We call this process of converting F# code to λ^H *Extraction*.

To start the process, it is necessary to access the abstract syntax tree of the source code. To achieve so, we use the F# Compiler Services (hereafter FSC) [13]. The FSC library is derived the F# compiler to provide developers with tools for working on the source code for F#, with identical behaviour to the actual F# compiler. In particular, it provides a representation of the typed F# abstract syntax tree (AST), which enables this work.

F# code is usually organised into a project, which often consists of a number of source files and possibly some external dependencies. We use project files as inputs to our library. We use the library `dotnet-proj-info` to analyse the project file and convert it to a form acceptable by FSC. FSC runs a typechecking pass with the F# type system and returns a typed AST of the source code for us to use. If there are type errors in the source code, our library terminates and reports these errors, since FLUIDTYPES always expects a well-typed program udner the F# type system.

Once the typed AST is obtained, we traverse the AST and follow a syntactic approach to convert the terms and types in the AST. For each project, the AST node consists of a list of files, where each file contains a list of definitions. The definitions are the focus of extraction, as the definitions, along with their types will be extracted.

4.2.1 Handling Expressions and Types

Expressions in the AST have type `FSharpExpr` and types have type `FSharpType`. Instead of being defined as a union type with a given number of cases to match against, the two data types are defined abstract, meaning the internals are not exposed, but FSC provides a number of *active patterns* for matching different syntactic constructs of F# programs. Active pattern is a language feature of F# that allows patterns to be defined and used in `match` expressions. We use mutually recursive functions to extract the F# types and expressions into λ^H terms and types, by pattern matching and then converting correspondent constructs in F# to λ^H.

Since F# has a wider range of language features and constructs than λ^H, we handle this issue in two ways. We implement extensions to the λ^H and its type system, and describe the extension in Section 4.4. For the rest of unsupported constructs, we define a special kind of terms `UnknownTerm` and similarly `UnknownType` for types. `UnknownType` is a type that is not supported in λ^H, and carries a string representation of the F# type. `UnknownTerm` similarly carries a string representation of the F# term, but also a type, which is extracted from the F# type. The type attached to an unknown term will be the type of the term in the type system. Refinement type annotation is not permitted in the case, since there is no way to check. These types and terms are treated as abstract, and we do not model any behaviour of them.

4.2.2 Summary

We described the `Extraction` module of FLUIDTYPES, responsible to converting F# expressions into λ^H terms. We convert common functional programming constructs (variable, function application, abstractions, conditional expressions, etc.) in F# into λ^H terms. While not all expressions are extractable, we handle these expressions in a graceful way. Later in Section 4.4, we introduce several extensions to the core features of the type system.

4.3 Annotations for Refinement Types

Our FLUIDTYPES library can extract terms and type information from F# code, which we discussed in Section 4.2. However, it would not be too interesting if there is no way to express type refinements, as all extracted types will have refinement `true` attached instead of any meaningful refinement types. We discuss the interface to interact with refinement types in the source code in this section.

The annotations are a crucial part of the library, since they form the interface for interacting with developers. We propose several approaches in this section. We first describe the chosen approach implemented, then discuss alternative approaches.

4.3.1 Annotation by Custom Attributes

Attribute is a feature in the F# language for attaching metadata [14]. They can be attached to a number of program constructs, and those metadata can be retrieved at runtime when necessary. For instance, the attribute marking the entry point of an executable is known as `[<EntryPoint>]` and we show an example usage in Figure 4.2.

```
1 [<EntryPoint>]
2 let main args =
3     Array.iter (printfn "%s") args
4     0
```

Figure 4.2: Code Snippet of Using an Attribute

Fortunately, it is possible to define custom attributes, so that we can use those attributes when imported. Moreover, attributes can carry some metadata so that they are not merely a tag as shown in the previous example. It is possible to attach basic data with the attributes, such as integers or strings. For instance, one can mark the deprecation of a function and provide an alternative, such as `[<Deprecated("foo")>]`.

We create a custom attribute `Refined` so that users can use the attribute to annotate refinement types for top-level definitions. The extraction process looks for any `Refined` attributes in the typed AST and extract refinement types out of the attribute metadata.

We discuss alternative approaches in Section 4.3.3, including some designs that are not possible to implement due to restrictions of runtime framework.

4.3.2 String-based Annotation

We use a string representation of the refinement type in the attribute, similar to the syntax of typed in Figure 3.1. Developers can use a string representation of the desired refinement type in the source code to annotate the top-level definitions. The string is then parsed by our typechecker, to an internal representation of λ^H. The syntax of refinement type is included in the appendix in Appendix C.1. Since not all terms can be encoded in SMT, as we previously discussed in Section 4.1.3, we do not support the full syntax of all λ^H terms in the syntax of refinement type annotations. In particular, we do not support abstractions and function applications.

Using this approach, we can annotate a function `abs` for the absolute value of an integer as in Figure 4.3. The return type is refined to be non-negative in the signature.

```
1 [<Refined("(x: int) → {v:int | v>=0}")>]
2 let abs x = if x > 0 then x else -x
```

Figure 4.3: `abs` Function with an Annotated Refinement Type

During the extraction process, the typechecker checks whether there exists a refinement type annotation. If there exists such an annotation, the annotation is parsed and checked against the F# type extracted from the typed AST. It is necessary that the annotated type from the string annotation be compatible to the extracted type from typed AST. The refinement type, after erasure, must be equal to the type extracted; in addition, the refinement type must be closed and well-formed, per well-formedness typing judgement. The term is checked against the annotated refinement type, if there exists one, or against the extracted unrefined F# type otherwise.

4.3.3 Alternative Designs

In this sub-section, we discuss alternative options of refinement annotation. We explain the benefits and drawbacks of each approach, and why they are not preferred over the approach chosen for the implementation.

4.3.3.1 Quotation-based approach

Code Quotation is a language feature of F# that allows a representation of the expressions of the language to be expressed in the language itself [15]. This provides a programmatic representation of F# expressions, with possibilities for typechecking.

We recall the refinement for base types is of form $\{v : b \mid M\}$, where M is a term. Here is where we can use a quoted expression instead of a string representation. In this

way, the well-formness of the refinement type can be guaranteed during compilation time by the F# typechecker, which saves our typechecker from performing typechecks.

The fundamental limitation of the approach is that attributes can only take literal values for metadata of base types. A quoted expression is unfortunately not among the supported literal values, so this approach is not feasible.

In addition, the requirement that the quoted expression must be closed may as well be a limitation. The term M in the refinement type is always closed, since it can contain the special variable ν along with variables in the typing context. The expressiveness of refinement types would be limited if only closed terms are permitted.

4.3.3.2 Special Comments

In LIQUIDHASKELL, the refinement type library for Haskell, the annotations are embedded in comments with a special form. The compiler still recognises the comment as a regular comment, but the typechecker is able to extract the information in the comment.

In a similar fashion, we can define a special form of comment for F# and extract information out of comments instead of using attributes. This approach lifts the restriction that F# attributes can only be attached to a limited number of program constructs.

The main drawback of this approach is that comments are not available in the typed AST after typechecking. Comments are usually discarded by compilers during the construction of the abstract syntax tree from the concrete parse tree, as they may not contain too much relevant information to later compilation stages after parsing. This approach is hence not implemented due to the engineering effort required, as it requires mapping information from the concrete syntax trees to typed abstract syntax trees manually.

4.3.4 Summary

We described the `Annotations` module of FLUIDTYPES, responsible for providing a way to annotate F# terms with refinement types. We discussed several design options for implementation, with their pros and cons, and implemented a string-based approach to annotate refinement types using F# attributes.

4.4 Type System Extensions

The core calculus λ^H does not include many features since the calculus is kept small and simplistic. In order to express the constraints necessary for typechecking refined protocols, and to improve the expressiveness of the type system, we implement a number of extensions and explain their details in this section.

4.4.1 Type Aliases

The limitations of F# attributes impose constraints on where the annotations can be added. To address this issue, we allow type aliases to be defined for refinement types, so values can be annotated in an explicit type annotation where a refinement attribute cannot be attached.

In particular, `let` expressions at a non-top-level cannot be annotated. For example, the code snippet shown in Figure 4.4 is invalid.

```
// error FS0824: Attributes are not permitted on 'let' bindings
// in expressions

let absAdd x y =
    let abs x: [<Refined("{v:int | v>=0}")>] int =
        if x >= 0 then x else -x
    abs x + abs y
```

Figure 4.4: Invalid Code Snippet due to a Limitation of F# Attributes

It is possible to attach a refinement type annotation on a type alias definition, so the type checker can check whether the refinement type is compatible to the aliased type, and remember the refinement type alias for future usages. For example, one can define a type alias for non-negative integers as shown in Figure 4.5.

```
[<Refined("{v:int | v>=0}")>]
type NonNeg = int
```

Figure 4.5: Type Alias Definition of Non-negative Integers

With the type alias definition, we can re-write the code in Figure 4.4 in a form that is accepted by F#, as shown in Figure 4.6.

```
let absAdd x y =
    let abs (x: int): NonNeg =
        if x >= 0 then x else -x
    abs x + abs y
```

Figure 4.6: Valid Code Snippet Using Type Alias

It is important to notice that type alias definitions are transparent, that is to say, the F# compiler does not distinguish between two types. We annotate explicit the argument `x` to function `abs` with type `int`, since F# would otherwise infer that `x` has type `NonNeg`, which may not be the intended type.

For implementation, we use a map to store the mapping between F# types and λ^H types. Originally, the map contains entried of builtin base types, such as base types for

boolean and integer. We add user-defined type aliases into the mapping. The refinement type annotation must be compatible to the aliased type and be well-formed, otherwise an error is reported.

4.4.2 Records

Records consist of a number of named fields, with their types explicitly stated in the type definition. The syntax of defining records is not too different from that in ML. A simple example is shown in Figure 4.7, where we define a record for a point on a plane.

```
type Point = {
    x : int;
    y : int;
}
```

Figure 4.7: An Example Definition of Record Type

In F#, field definitions can carry attributes, so we can utilise this feature to add refinement annotations to fields, such as in Figure 4.8. We define a record for a point in the first quadrant, so both the x and y value of the point are positive, as specified by the refinement type annotation.

```
type PointFirstQuadrant = {
    [<Refined("{v:int | v>0}")>] x : int;
    [<Refined("{v:int | v>0}")>] y : int;
}
```

Figure 4.8: An Example Definition of Record Type with Refinement

Moreover, subsequent field declarations can refer to previously defined fields in the refinement type annotation. In this way, it is possible to specify data dependencies in records, and this gives us a way to specify a dependent record type in this fashion. We can encode existential types in our annotated record type. An example is to encode a type for even numbers, as shown in Figure 4.9, where `num` is the even number, and `half` provides the evidence that the even number is twice of `half`. With this definition, we can prove that the sum of two even numbers are even, as shown in Figure 4.10.

```
type EvenNumber = {
    half : int;
    [<Refined("{v:int | v=half+half}")>] num : int;
}
```

Figure 4.9: An Example Definition of Record Type with Data Dependency

```
let evenPlus (x: EvenNumber) (y: EvenNumber): EvenNumber =
    {
        half = x.half + y.half
        num = x.num + y.num
    }
```

Figure 4.10: Proving Sum of Two Even Numbers are Even Using Record Type

To check whether a definition of record with refinement is well-formed, we check the field definitions one by one. We use an empty typing context to begin with, and then append field variables with their types. The refinement attached on each field definition must be compatible with their F# type and that the refinement type must be well-formed under the current typing context. A refinement type may optionally refer to a field that is defined prior to that field, since the fields defined earlier has entried in the typing context with their types.

If a user-defined record is well-formed, it is added to a record definition environment. When a new record of a user-defined type is created, we retrieve the definition from the environment, and check each field according to the order they are defined. Moreover, fields form equality predicates that are added to the predicate context. This allows the data dependencies within a record to be expressed.

For encoding of the records into SMT logic, we currently flatten the records into a number of variables, which corresponds to the fields. This approach imposes limitations on the expressiveness, as a record must be bound to a variable, in order for the data dependencies to be correctly encoded.

Alternatively, SMT logic supports custom data type definitions, which can encode sum and product data types into the logic. This alternative approach can be implemented as an extension to this work.

4.4.3 Enumerations

Enumerations, or enums for short, provide a means to assign labels to values. In F#, the values are limited to integers only. Therefore, during compilation, the enum values are de-sugared into integer values in the typed AST, except that they carry the type of the enum instead. We use this feature in FLUIDSESSION to express the branching behaviour of a protocol.

An example enum definition is shown in Figure 4.11. This enum `MyEnum` has two values, `Apple` with value `1` and `Pineapple` with value `3`.

In FLUIDTYPES, we consider enums as a type alias to unrefined integers. The values of the enums available in the typed AST are already de-sugared integer values, so they always typecheck against the unrefined integers. A more precise type for enums is integers refined to only take the set of possible values as declared in the definition. For example, `MyEnum` would be a alias to $\{\nu : \texttt{int} \mid \nu = 1 \vee \nu = 3\}$, instead of unrefined

```
type MyEnum =
  | Apple = 1
  | Pineapple = 3
```

Figure 4.11: An Example Type Definition of Enumeration Type

`int`. Since it is not possible to construct an integral value outside those mapped to by a label in F#, we take advantage of this guarantee and safely use the unrefined `int` type.

4.4.4 Algebraic Data Types

FLUIDTYPES has support for discriminated unions and limited support for tuples.

For discriminated unions, it is possible to add refinement type annotation for each union case. For example, we can represent integers according to their signs with the union type defined in Figure 4.12

```
type Number =
    | [<Refined("{v:int | v>0}")>] Pos of int
    | [<Refined("{v:int | v=0}")>] Zero of int
    | [<Refined("{v:int | v<0}")>] Neg of int
```

Figure 4.12: An Example Type Definition of Discriminated Union with Refinements

For each union case in the union type definition, we generate refinement-aware function signatures in the typing context for constructors, case testers (which checks whether a union value matches a given tag) and eliminators. For example, the `Pos` case has a constructor of type $(v : \{\nu : \texttt{int} \mid \nu > 0\}) \rightarrow \texttt{Number}$, a case testing function of type $\texttt{Number} \rightarrow \texttt{bool}$, and an eliminator of type $\texttt{Number} \rightarrow \{\nu : \texttt{int} \mid \nu > 0\}$.

In F#, pattern matching does not strictly follow the traditional formalisation of a *match* constructs, where each case of the union is provided with a expression to execute when the tag matches. Instead, the `match` expression is de-sugared into a series of conditional expressions with case testers as conditions, and then use eliminators to obtain the tagged value. Therefore, we can support discriminated unions with λ^H with the generated function definitions.

For tuples, the support is very limited. FLUIDTYPES is able to extract tuples used in F#, but there is no way to specify refinement type annotations on them. This can be addressed in future work. Despite that tuples with refinements cannot be expressed now, it is possible to use records to achieve a similar effect.

Chapter 5

FLUIDSESSION: Towards Statically Verified Protocol Implementation

In this chapter, we present a way to use FLUIDTYPES to statically verify communication protocol implementation. We present a code generator in F# to provide APIs with refinement types for protocols written in the Scribble protocol description language. Developers can implement the protocol using the provided APIs, and then use FLUIDTYPES to check statically whether the protocol implementations are correct according to the refinements. The code generator is publicly available under the MIT License at `https://github.com/fangyi-zhou/ScribbleCodeGen`.

5.1 Protocol Specification with Scribble

We use a variant of the Scribble protocol description language, as presented in [36]. This variant allows message payload to be named with variables, and assertions can then be attached to these messages. Details of the modified Scribble are previously introduced in Section 2.2.4.

5.2 Obtaining the CFSM from Scribble

As in previous work [36], code generation uses the Scribble toolchain to extract information about the protocol. Scribble gives a local projection for a given role of a global protocol in the form of a communicating finite state machine (hereafter CFSM).

We first invoke Scribble to obtain to graph representation of CFSM for the given role to begin with. Transitions of the CFSM are in the form of communication actions (i.e. sending or receiving messages to another role). Scribble first validates the well-formedness of the protocol, and then provides the output in the DOT graph description language, with communication actions encoded as a string attached to state transitions. Our code generator parses the graph representation, including the action labels into an internal representation of CFSM.

5.3 API Generation

We explain the process of API generation in this section. In prior work [36], APIs are presented in a way similar to Java APIs, as previously described in Section 2.1.4. This approach can provide safety guarantees, but only when all the state objects are used in a linear fashion, which requires careful attention by the developers.

We use an event-based approach of API generation in contrast to the style used in previous work. We motivate this handler style from the idea originally proposed by Hu in [25]. This style specifies handlers for send and receive actions, instead of exposing send and receive APIs to users. When all handlers are specified, the CFSM can be executed as a deterministic automata, which guarantees linearity by construction.

We use a running example to demonstrate how the API generation works in our work. To begin with, we use a simple Adder protocol involving two roles, a Server and a Client, as shown in Figure 5.1, and gradually we evolve the example with more features.

5.3.1 A Straight-line Protocol without Refinements

```
1 global protocol Adder(role C, role S) {
2     NUM1(x: int) from C to S;
3     NUM2(y: int) from C to S;
4     SUM (z: int) from S to C;
5 }
```

Figure 5.1: Adder Protocol (without Refinements)

The protocol is quite simple. The client sends two messages to the server, `NUM1` and `NUM2` each containing an integer. After that, the server will send back an integer in a `SUM` message, for the sum of the two values received from the client. Currently, the protocol does not have refinements, but we will add refinements in later examples.

We obtain a CFSM from Scribble for role Client, and show it in Figure 5.2. Since there are no branches or recursions, the CFSM is in a form of a straight line.

We generate a type for each state in the CFSM, which we explain more in Section 5.3.3. For send actions in transitions, the handlers are functions that take the origin state as an argument, and return a value with the type of the payload. For example, the handler for the send transition from state 6 to state 8 has the following type:

```
1 state6OnsendNUM1 : State6 → int
```

This handler will be called during execution at state 6. The return value of the function provides the payload value to send in `NUM1` message. After that, the CFSM progresses to the next state.

For receive actions, the handlers are functions that take the origin state and the payloads as arguments, and return an unit. For example, the handler for the receive

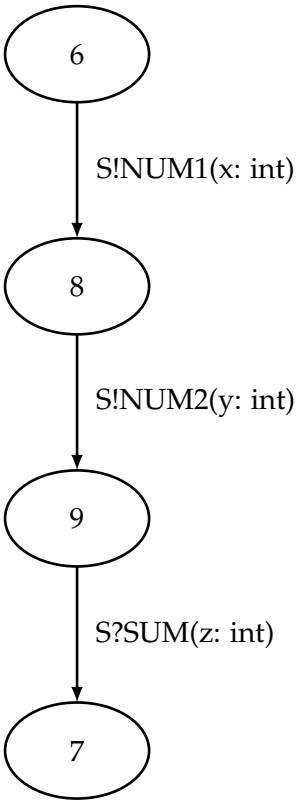

Figure 5.2: Finite State Machine for Role C for Protocol in Figure 5.1

transition from state 9 to state 7 has the following type:

```
state9OnreceiveSUM : State9 → int → unit
```

This handler will be called during execution at state 9. The payload value received in a SUM message is passed as an argument of the handler function. After that, the CFSM progresses to the next state.

The handlers are gathered in a record, with its type definition shown in Figure 5.3. Developers only need to provide the handlers to implement the protocol. Once the handlers are provided, the execution of CFSM is through generated code, so developers do not need to maintain states manually. We explain how the execution of CFSM works in Section 5.4.

```
type HandlersC = {
    state6OnsendNUM1 : State6 → int
    state8OnsendNUM2 : State8 → int
    state9OnreceiveSUM : State9 → int → unit
}
```

Figure 5.3: Generated Handler Types for role C for Figure 5.1

5.3.2 Adding Refinements to Payloads

In the next example, shown in Figure 5.4, we attach refinements to the messages, so that the integers being exchanged must be non-negative. In Scribble, the assertions are attached to messages, whereas in refinement types, the refinements are attached to types. As a consequence, it is necessary to convert the assertion attached to the message, to refinement types of the payload.

In the current implementation, the payload can carry exactly one value, therefore attaching refinement to type is trivial. The limitation arises from the absence of support of algebraic data types in FLUIDTYPES. However, we do not lose generality in this case, since a message with multiple payloads can be decomposed into a series of messages with a single payload.

```
1 global protocol Adder(role C, role S) {
2     NUM1(x: int) from C to S; @"x >= 0"
3     NUM2(y: int) from C to S; @"y >= 0"
4     SUM (z: int) from S to C; @"z >= 0"
5 }
```

Figure 5.4: Adder Protocol (with Refinements)

The handlers are now annotated with refinement type attributes as described in Section 4.3. The type definitions of all the handlers are shown in Figure 5.5. We notice the return type of the send handlers are refined to be non-negative, so the implementation must provide a function satisfying the refinement. Similarly, the receive handler has a non-negative in the argument type, so that the argument can carry the non-negative refinement when being used in the handler definition.

```
1 type HandlersC = {
2     [<Refined("State6 → {x:int | x>=0}")>]
3     state6OnsendNUM1 : State6 → int
4     [<Refined("State8 → {y:int | y>=0}")>]
5     state8OnsendNUM2 : State8 → int
6     [<Refined("State9 → (z: {z:int | z>=0}) → unit")>]
7     state9OnreceiveSUM : State9 → int → unit
8 }
```

Figure 5.5: Generated Handler Types for role C for Figure 5.4

5.3.3 Adding Refinements with Non-payload Variables

In the third example, shown in Figure 5.6, we add another refinement to the SUM message, that the payload must be the sum of the two integers in the previous messages. Previously in Figure 5.4, all variables in the refinement are those defined in the payload.

In the new example, the refinement also includes variables that are known by the local role, but not part of the payload itself. For example, the message SUM has refinement $z = x + y$, where x and y are known via previous messages, and z is a payload variable. Note that Scribble validates that all variables in the refinements are known by the sending role of the message.

```
global protocol Adder(role C, role S) {
    NUM1(x: int) from C to S; @"x >= 0"
    NUM2(y: int) from C to S; @"y >= 0"
    SUM (z: int) from S to C; @"z >= 0 && z = x + y"
}
```

Figure 5.6: Adder Protocol (with More Refinements)

This means a naive generation, as shown below, would not be valid, since the type is not well-formed due to unbound variables x and y:

```
[<Refined("State9 → (z: {z:int | z>=0 && z=x+y}) → unit")>]
state9OnreceiveSUM : State9 → int → unit
```

To fix this issue, we need to bind the variables x and y in an appropriate context, while not breaking validity in the meantime. It is crucial for correctness that these variables are bound to be the exact values received in previous messages. Therefore, we need to place the variables x and y in the state type State9.

Therefore, for each state, we generate a record type containing all the variable known at the state, including their refinements. Fortunately, we are able to express dependencies between fields, as this is implemented by the FLUIDTYPES record extension (described in Section 4.4.2). We traverse the CFSM states in a depth-first way, and collect the variables defined in the payload in the messages and their refinements. The record for State9 contains the variables x and y, as shown in Figure 5.7

```
type State9 = {
    [<Refined("{x:int | x>=0}")>]
    x : int
    [<Refined("{y:int | y>=0}")>]
    y : int
}
```

Figure 5.7: Generated State Record Definition for State 9

With the state record containing a backlog of previously exchanged messages, the handler can have access to the necessary variables. These records will be constructed by the execution function at runtime, We can thus give the following refinement type to the send handler, with variables x and y bound to the state record type ($ operator is used for field access in records):

```
[<Refined(
    "(st: State9) → (z: {z:int | z>=0 && z=st$x+st$y}) → unit"
)>]
state9OnreceiveSUM : State9 → int → unit
```

5.3.4 Adding Branches

In the fourth example, shown in Figure 5.8, we extend server with the ability to compare the operands of addition. The result of the comparison is identified by the label of the message sent from the server. `GEQ` means the first number is greater than or equal to the second one, `LES` means otherwise. The refinements attached to `GEQ` and `LES` message ensure that the message are only sent if the correct comparison holds.

```
global protocol Adder(role C, role S) {
    NUM1 (x: int) from C to S; @"x >= 0"
    NUM2 (y: int) from C to S; @"y >= 0"
    choice at S {
        GEQ (z1: int) from S to C;
            @"(z1 >= 0 && z1 = x + y) && x >= y"
    } or {
        LES (z2: int) from S to C;
            @"(z2 >= 0 && z2 = x + y) && x < y"
    }
}
```

Figure 5.8: Adder Protocol (with Branches)

The CFSM for the client changes due to the branching behaviour. We show the new CFSM in Figure 5.9. Notice that there are two transitions from state 9 to state 7 in the CFSM, which corresponds to the two branches after the choice.

Recall that we generate handlers for each receive action. In this protocol, the handler generation follows a similar process. We generate the following types for handlers:

```
[<Refined(
    "(st: State9)
   → (z1: {z1:int | z1>=0 && z1=st$x+st$y && st$x>=st$y})
   → unit"
)>]
```

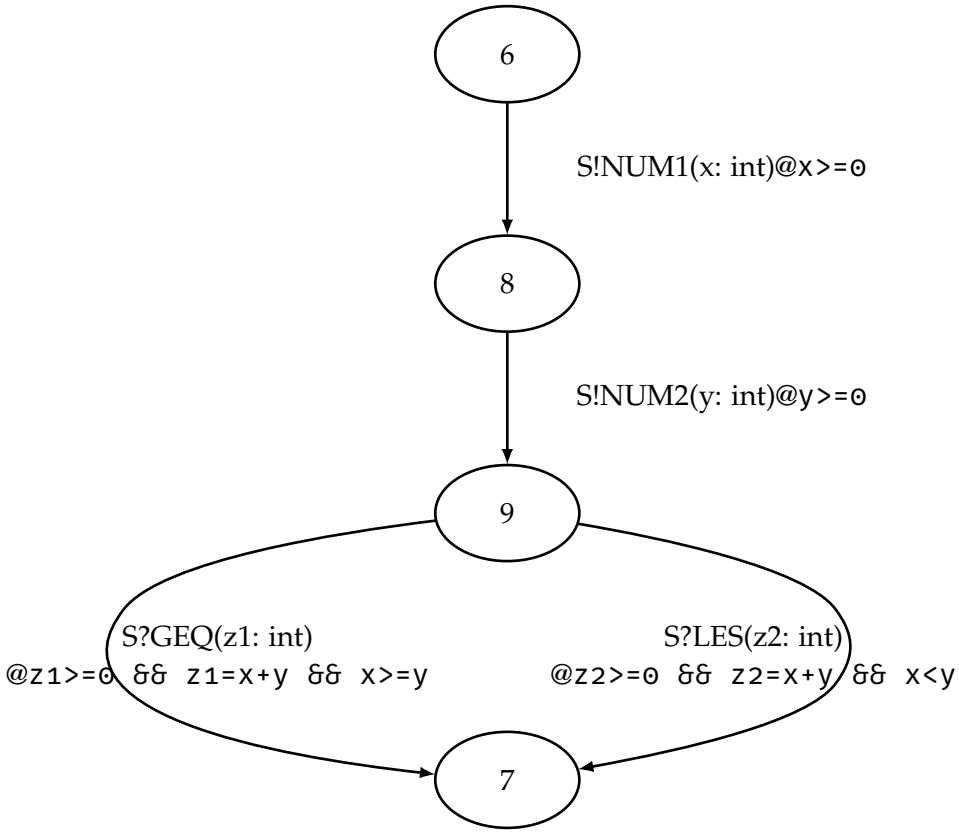

Figure 5.9: Finite State Machine for role C for Protocol in Figure 5.8

```
state9OnreceiveGEQ : State9 → int → unit
[<Refined(
    "(st: State9)
   → (z2: {z2:int | z2>=0 && z2=st$x+st$y && st$x<st$y})
   → unit"
)>]
state9OnreceiveLES : State9 → int → unit
```

The label of the message is received first during the execution at state 9, following the appropriate payload of the message according to the label. Then, the appropriate receive handler is invoked, with the received payload. After that, the CFSM progresses to the next state.

On the other hand, API generation for Server side cannot follow the same approach. Since there are only two parties in this protocol, the CFSM for S has the same states as that for C, but the actions of the state transitions are swapped between send actions and receive actions.

Following the same approach, we would generate the APIs for transition from state 9 to state 7 as follows:

```
// This API generation is incorrect
[<Refined(
    "(st: State9) → {z1:int | z1>=0 && z1=st$x+st$y && st$x>=st$y}"
)>]
state9OnsendGEQ : State9 → int
[<Refined(
    "(st: State9) → {z2:int | z2>=0 && z2=st$x+st$y && st$x<st$y}"
)>]
state9OnsendLES : State9 → int
```

At state 9, it is not clear which handler should be invoked for the CFSM to progress. According to the protocol, this is a choice at Server. It is necessary to provide another handler for selecting which handler to invoke. As a consequence, for a state with multiple outward send transitions in a CFSM, we generate an enum for each label, and a handler for selecting which label to progress with when there is a choice. This gives us the following APIs:

```
type State9Choice =
    | GEQ = 0
    | LES = 1
state9 : State9 → State9Choice
[<Refined(
    "(st: State9) → {z1:int | z1>=0 && z1=st$x+st$y && st$x>=st$y}"
)>]
state9OnsendGEQ : State9 → int
[<Refined(
    "(st: State9) → {z2:int | z2>=0 && z2=st$x+st$y && st$x<st$y}"
)>]
state9OnsendLES : State9 → int
```

In this style, the handler for `state9` is invoked first, where the return value specifies the label to continue with protocol with. Then, the appropriate send handler is invoked, where the return value can be communicated as the payload of the message.

However, this approach has drawbacks. Recall the message `GEQ` has the refinements $z1 \geq 0$, $z1 = x + y$ and $x \geq y$, among which the refinement $x \geq y$ does not involve the variable of the payload $z1$. This is a constraint on branching behaviour and such constraint is not reflected by the refinement type of label selector `state9`. Furthermore, we suppose that the label selector is implemented in a way that satisfies the constraints attached to each label, but the send handler for the label does not take those constraints into account in state 9. That is to say, the developer may need to perform redundant checks to satisfy the refinement type system, which is inefficient.

To address this problem, we refine the label selection handler. For each send action, we split the refinements into two parts, depending on whether the refinement contains payload variables. We call the part without payload variables *path constraints*. The

return type of label selector is refined such that a label can only be selected if the path constraint for that label is satisfied. Practically, this means the refinement of the label is a disjunction of refinements for each label, where the refinement for each label is the conjunction of the path constraints and the selection of label as the return value.

In this example, `GEQ` has path constraint $x \geq y$; `LES` has path constraint $x < y$. Hence the refinement for `GEQ` label is `(choice = 0) && (st$x >= st$y)`; that for `LES` label is `(choice = 1) && (st$x < st$y)`, where `st` refers to a record of type `State9`. Therefore, we have the following refined type for `state9` handler.

```
[<Refined(
    "(st: State9)
  → {label:int | (label=0 && st$x>=st$y) || (label=1 && st$x<st$y)}"
)>]
state9 : State9 → State9Choice
```

For the send handlers, we know that the path constraints must have been satisfied from the refined label selection handler. Hence the path constraints need to be added to the record definition of the state. We generate new record definitions for state 9 for each label with path constraints (here only showing `GEQ`). Notice that `y` is refined such that $x \geq y$:

```
type State9_GEQ = {
    [<Refined("{x:int | x>=0}")>
    x : int
    [<Refined("{y:int | y>=0 && x>=y}")>]
    y : int
}
```

We generate send handlers using the records types with path constraints, as shown in the following snippet:

```
[<Refined(
    "(st: State9_GEQ)
  → {z1:int | z1>=0 && z1=st$x+st$y && st$x>=st$y}"
)>]
state9OnsendGEQ : State9_GEQ → int
```

While it may be tedious to use a label section handler first and then handlers for send actions, it is possible to use one handler for both. This comes at the cost that no refinement can be expressed in the message. We could generate `State9Choice` as a discriminated union with payloads. However, it is not possible in the current setup of FLUIDTYPES, to express dependencies of the discrimated union with regard to the state. This could be added if F# provides type families, so we can use a state type to index into the choice type and establish correct data dependencies.

5.3.5 Summary

We summarise the generation of handlers in Table 5.1 according the outward transitions of a state.

Action	Number	Generation
Send	One	1 send handler
	Multiple	1 label selector and 1 send handler for each label
Receive	Any	1 receive handler for each label

Table 5.1: Handler Generation for a State in CFSM

For each non-terminal state of the CFSM, we create a record to store all variables known at that state since the initial state, with their refinement types. For states with multiple outgoing send actions, we find the path constraints for each action, and create a record with additional path constraints.

5.4 Execution of the CFSM

We outline the code generation procedure for the execution of CFSM. We use a `run` function to execute the CFSM and communicate with other roles, after all handlers are provided by the developer. Due to limited time, this part is currently not implemented in FLUIDSESSION. However, the limitation that implementations cannot be executed does not invalidate the result of verification, since verification is done by typechecking the handlers.

The execution of CFSM takes a record of handlers provided by the developer, and has three major aspects:

- Perform communication to other roles:

  The execution function establishes connections and exchange messages with other roles. It serialises and de-serialises the payload and transmits the content to other roles. Therefore, the developers do not need to handle tedious communication code.

  Since all communication are done completely by the execution function, the CFSM keeps track of the usage of communication channels. In this way, we can ensure the linearity of the channels easily. This is a major advantage of this style of API generation.

- Invoke correct handlers according to the current state:

  The execution function remembers the current state and invoke appropriate handlers before or after the communication.

For single outgoing send actions, the send handler for the appropriate state is invoked, and the return value from the handler is sent to the receiver role, along with the label.

For multiple outgoing send actions, the label selection handler for the appropriate state is invoked, and then the send handler corresponding to the return value is invoked. The return value from the send handler is sent to the receiver role, along with the label.

For receive actions, the label is received first, then the correspondent payload with the label. The appropriate receive handler is then invoked with the payload.

- Construct state records:

  The execution function keeps the backlog of the history of communication in state records. When communication takes place and state changes, the function constructs a new state record for the next handler to use.

5.5 Limitations

Message payload is limited to one item in the current setup, due to limited support for tuple types in FLUIDTYPES; However, it is possible to generate records in the case of the multiple payload items to address this limitation. We do not lose generality due to this limitation.

It is not possible to express refinements in different iterations of recursion in a protocol. Allowing such refinements would give a huge boost to expressiveness, as invariants can be established in this way. This limitation originates from the previous work of [36], since we use the same version of Scribble in our work, we also face the same limitation. This is an open research question, and we discuss possible improvements in Section 6.3.

Chapter 6

Evaluation

The main objective of our work is to statically verify implementations of protocols with refinements according to their specification. We use three cases studies of protocols to demonstrate strengths and weaknesses of our work:

- Adder Protocol in Section 5.3:

  We walk through an implementation of the Adder Protocol, where a client asks a server to perform some simple computation of addition and comparison. This protocol contains the main features of protocols in Scribble. We demonstrate the interaction of FLUIDSESSION and FLUIDTYPES via examples.

- Two Buyers Protocol:

  We use a variant of two buyers protocol from [23], involving 3 participants. This illustrates that our project is not limited to binary communications protocols, but that it also extends to multiple parties.

- Accumulator Protocol:

  We demonstrate limitations of our work via an accumulator protocol between two parties, where a series of values are sent from the client to the server, and the server accumulates the values by addition. The refinements in the protocol cannot be expressed by our current implementation. However, we propose possible future works to address the limitation. The limitation arises from the Scribble toolchain, instead of our approach to generate code.

6.1 Adder Example in Section 5.3

We recall the protocol definition of the Adder protocol, as shown in Figure 5.8. The Client sends two non-negative integers to the Server for some computation, and the Server sends back the comparison result and their sum.

```
global protocol Adder(role C, role S) {
    NUM1 (x: int) from C to S; @"x >= 0"
    NUM2 (y: int) from C to S; @"y >= 0"
    choice at S {
        GEQ (z1: int) from S to C;
            @"(z1 >= 0 && z1 = x + y) && x >= y"
    } or {
        LES (z2: int) from S to C;
            @"(z2 >= 0 && z2 = x + y) && x < y"
    }
}
```

Figure 5.8: Adder Protocol (repeated from page 68)

6.1.1 Client

We project the global protocol into a CFSM representation for Client, and use our code generator to generated APIs with refinement types. For the Client side, the handler type is shown in Figure 6.1.

```
type HandlerC = {
    [<Refined("State6 → {x:int | x>=0}")>]
    state6OnsendNUM1 : State6 → int
    [<Refined("State8 → {y:int | y>=0}")>]
    state8OnsendNUM2 : State8 → int
    [<Refined(
        "(st: State9)
       → (z1: {z1:int | z1>=0 && z1=st$x+st$y && st$x>=st$y})
       → unit"
    )>]
    state9OnreceiveGEQ : State9 → int → unit
    [<Refined(
        "(st: State9)
       → (z2: {z2:int | z2>=0 && z2=st$x+st$y && st$x<st$y})
       → unit"
    )>]
    state9OnreceiveLES : State9 → int → unit
}
```

Figure 6.1: Handler Type Definition for Client in Adder Protocol

In this case, a developer needs to implement 4 handlers for the Client role. The first two send handlers ask the programmer to provide two numbers to be added together, to be sent to the Server. The other two receive handlers ask the programmer to process received sum value. We provide a simple example implementation in Figure 6.2.

```
module ClientImpl

// Import Generated Code
open ScribbleGeneratedAdderC

let handlers = {
    state6OnsendNUM1
        = fun _ → 1    // Send integer 1
    state8OnsendNUM2
        = fun _ → 2    // Send integer 2
    state9OnreceiveGEQ
        = fun _ _ → () // Do nothing
    state9OnreceiveLES
        = fun _ _ → () // Do nothing
}
```

Figure 6.2: Example Implementation of Client Role

In the example implementation, we use integer `1` for message `NUM1` and integer `2` for message `NUM2`. These numbers are surely non-negative and pass the typecheck. For handling receive actions, we simply do nothing.

This simple implementation passes the refinement typecheck by FLUIDTYPES:

```
$ mono FluidTypesConsole.exe Client.fsproj
All checks passed!
```

However, if we replace integer `1` in message `NUM1` with `-1`, FLUIDTYPES fails the with following error message:

```
$ mono FluidTypesConsole.exe Client.fsproj
File Adder/ClientImpl.fs
BaseType (TInt,
          App (App (Const (Binop EqualInt), Var "$this"),
               Const (IntLiteral -1)))
is not a subtype of
BaseType (TInt,
          App (App (Const (Binop GreaterEqual), Var "$this"),
               Const (IntLiteral 0)))
for term
Const (IntLiteral -1)
```

We use an internal representation of λ^H terms and types in the error message. This error means that the term -1 has type $\{\nu : \texttt{int} \mid \nu = -1\}$, which is not a subtype of $\{\nu : \texttt{int} \mid \nu \geq 0\}$. For the handler to be correctly typed, the return value for the send handler must be a non-negative integer. As expected, FLUIDTYPES reports an error.

For a more sophisticated implementation, we can use a function to generate a random

integer as a return value of the handler. We can define a function as follows, using the builtin random number generator provided by .NET APIs:

```
let randomInt () = System.Random().Next()
```

We can use this function in the send handlers:

```
state6OnsendNUM1 = fun _ → randomInt ()
```

However, since there is no domain modelling for .NET APIs in FLUIDTYPES, it is not possible to obtain more information other than the return value is an integer. Therefore, FLUIDTYPES produces the following error message:

```
$ mono FluidTypesConsole.exe Client.fsproj
File Adder/ClientImpl.fs
BaseType (TInt, Const (BoolLiteral true))
is not a subtype of
BaseType (TInt,
          App (App (Const (Binop GreaterEqual), Var "$this"),
               Const (IntLiteral 0)))
for term
UnknownTerm
    ("Call (None, val randomInt, [], [],
            [Const (null,type Microsoft.FSharp.Core.unit)])",
     BaseType (TInt, Const (BoolLiteral true)))
```

To address this issue, we introduce an extra check to refine the type, and negate the random value when it is negative:

```
state6OnsendNUM1 = fun _ →
    let rnd = randomInt ()
    if rnd >= 0 then rnd else -rnd
```

With the changes, the implementation passes the typecheck of FLUIDTYPES:

```
$ mono FluidTypesConsole.exe Client.fsproj
All checks passed!
```

To check the properties of the received value, we first define some auxiliary functions in Figure 6.3. `expect_sum` takes 3 integers, where the third argument is expected to be the sum of the first 2 integer arguments. `expect_nonneg` takes an integer that is non-negative. `expect_pos` takes an integer that is positive. We can call these auxiliary functions, and FLUIDTYPES checks whether their arguments carry the types specified in the refinement annotations. We use them to test properties about the received value in the receive handler.

We can check whether the received value is the sum of the two numbers as follows:

```
state9OnreceiveGEQ = fun st value → expect_sum st.x st.y value
```

```
[<Refined("(x: int) → (y: int) → (sum: {v:int | v=x+y}) → unit")>]
let expect_sum x y sum = ()

[<Refined("(x: {v:int | v>=0}) → unit")]
let expect_nonneg x = ()

[<Refined("(x: {v:int | v>0}) → unit")]
let expect_pos x = ()
```

Figure 6.3: Auxiliary Functions for Testing Properties

Similarly, we can validate that after receiving `LES`, we know $x \leq y$. To validate this property, we check whether $y - x$ is positive:

```
state9OnreceiveLES = fun st _ → expect_pos (st.y - st.x)
```

We validate the properties by invoking FLUIDTYPES again:

```
$ mono FluidTypesConsole.exe Client.fsproj
All checks passed!
```

We validate these properties successfully by FLUIDTYPES with static typechecking. This can save developers from writing unit tests or dynamic checks to ensure those properties hold, because they can be checked statically during compile time.

It is important to notice that FLUIDTYPES checks refinement according to the refinements specified in the protocol specification. A possible implementation may choose only to send positive integers in `NUM1` and `NUM2`, as shown in Figure 6.4 and expect the sum to be always positive. These properties cannot be validated by FLUIDTYPES, because our work aims to provide guarantees according to the *protocol description*, but not generic properties established by a *specific implementation*, that are not required by the protocol.

FLUIDTYPES fails with an error at the argument of `expect_pos` in the handler for `GEQ` message. Readers may wonder why no error is produced in the handler for `LES` message. The reason is that the SMT solver used by FLUIDTYPES is able to prove semantic properties. In this case, we have that x and y are non-negative, and that $x < y$, we know that it must be the case that $y > 0$. Therefore, $x + y$ must be positive and the desired validity holds.

In the case of `GEQ` message, the solver cannot prove that $x + y > 0$, since $x = 0$ and $y = 0$ are able to satisfy the necessary constraints $x \geq y$ imposed by refinement on the message, but does not satisfy $x + y > 0$. Since there exists a counterexample, the desired validity does not hold. FLUIDTYPES hence produces an error:

```
let handlers = {
    state6OnsendNUM1
        = fun _ →
            let rnd = randomInt ()       // The value sent to the
            if rnd > 0 then rnd else 42  // server is always positive
    state8OnsendNUM2
        = fun _ →
            let rnd = randomInt ()       // The value sent to the
            if rnd > 0 then rnd else 42  // server is always positive
    state9OnreceiveGEQ
        = fun _ value →
            expect_pos value             // Error here
    state9OnreceiveLES
        = fun _ value →
            expect_pos value             // No error here
}
```

Figure 6.4: Alternative Implementation of Client Role

```
$ mono FluidTypesConsole.exe Client.fsproj
File Adder/ClientImpl.fs
BaseType (TInt,
          App (App (Const (Binop EqualInt), Var "$this"),
               Var "value"))
is not a subtype of
BaseType (TInt,
          App (App (Const (Binop Greater), Var "$this"),
               Const (IntLiteral 0)))
for term
Var "value"
```

FLUIDTYPES is not designed to prove properties about a specific implementation of the protocol, but instead to prove properties *specified* in the protocol. In this example, the implementation chooses to make the additional guarantee that all numbers are positive, whereas the protocol only requires non-negative. Such implementation-defined properties require a more sophisticated analysis with knowledge of execution flow, and proving them is a non-goal for our design.

6.1.2 Server

We project the global protocol into a CFSM representation for Server, and use our code generator to generated APIs with refinement types. For the Server side, the handler type is shown in Figure 6.5.

In this case, a developer needs to implement 5 handlers. There are 2 receive handlers for the operand numbers, 1 handler to select the label for the next send action and 2 send

```
type HandlerS = {
    [<Refined("State14 → (x: {x:int | x>=0}) → unit")>]
    state14OnreceiveNUM1 : State14 → int → unit
    [<Refined("State16 → (y: {y:int | y>=0}) → unit")>]
    state16OnreceiveNUM2 : State16 → int → unit
    [<Refined(
        "(st: State17)
      → {label:int |
            (label = 0 && st$x < st$y)
          || (label = 1 && st$x >= st$y)
       }"
    )>]
    state17 : State17 → State17Choice
    [<Refined(
        "(st: State17_LES)
      → {z2:int | z2>=0 && z2=st$x+st$y && st$x<st$y}"
    )>]
    state17OnsendLES : State17_LES → int
    [<Refined(
        "(st: State17_GEQ)
      → {z1:int | z1>=0 && z1=st$x+st$y && st$x>=st$y}"
    )>]
    state17OnsendGEQ : State17_GEQ → int
}
```

Figure 6.5: Handler Type Definition for Server in Adder Protocol

handlers. We provide a simple example implementation in Figure 6.6.

In the minimal implementation, we handle receive actions by doing nothing. There is no need to save the received values explicitly, because they are stored in the state record automatically by the execution function. We compare the values of two received values to decide which action to proceed in `state17`. In the send handlers for both `LES` and `GEQ`, we add the two received values together.

This minimal implementation of the handlers passes the refinement typecheck by FLUIDTYPES:

```
$ mono FluidTypesConsole.exe Server.fsproj
All checks passed!
```

We can check properties on the received values, similar to what we previously did for the Client role handlers. We can use `expect_nonneg` function in the receive handler to ask FLUIDTYPES to validate that the number received is non-negative. We do not demonstrate similar checks here to avoid duplication.

Instead, we focus on the label selection handler in this implementation. A careless programmer may drop the equal sign at the comparison, and cause a protocol violation in the edge cases. We use an incorrect implementation of the handler `state17` , shown

```
module ServerImpl

// Import Generated Code
open ScribbleGeneratedAdderS

let handlers = {
    state14OnreceiveNUM1
        = fun _ _ → ()                  // Do nothing here
    state16OnreceiveNUM2
        = fun _ _ → ()                  // Do nothing here
    state17
        = fun st →
            if st.x >= st.y
            then State17Choice.GEQ      // GEQ when x >= y
            else State17Choice.LES      // LES otherwise
    state17OnsendLES
        = fun st → st.x + st.y          // Adding values together
    state17OnsendGEQ
        = fun st → st.x + st.y          // Adding values together
}
```

Figure 6.6: Example Implementation of Server Role

as follows:

```
state17 = fun st →
    if st.x > st.y          // `>` is used instead of `>=`
    then State17Choice.GEQ  // no error, since x > y implies x >= y
    else State17Choice.LES  // error when x = y
```

Running FLUIDTYPES produces an error as expected. The error message indicates that `0`, the enum value for `LES`, does not fulfill the refinement constraints. This is because $x \leq y$ does not hold when $x = 0$ and $y = 0$, but such values leads to the `LES` label being selected incorrectly.

```
mono FluidTypesConsole.exe Server.fsproj
File Adder/ServerImpl.fs
BaseType (TInt,
          App (App (Const (Binop EqualInt), Var "$this"),
               Const (IntLiteral 0)))
is not a subtype of
BaseType
  (TInt,
   App ...
for term
Const (IntLiteral 0)
```

6.1.3 Summary

We demonstrated how to implement a simple binary protocol using generated APIs from FLUIDSESSION, and use FLUIDTYPES to check the correctness of implementation with regards to the specification. The `Adder` protocol covers main features of Scribble, including messages and branches, along with refinements. We showed how FLUIDTYPES can check static properties and catch common implementation errors through examples.

The error messages produced by FLUIDTYPES are not friendly to developers in their current form, as they use an internal representation of terms instead of the F# source code. An aspect for future work is to make relevant connections between extracted terms and original source code locations, to improve the usability of the library.

6.2 Two Buyers Protocol

We implement a variant of the two buyers protocol from [23], with refinements. The two buyer protocol describes a protocol between two buyers A, B and a Seller. Through this example, we show that our approach extends to multiparty scenarios.

We show the Scribble specification of this protocol in Figure 6.7. A and B are two buyers, and S is the seller. A asks the Seller for a quote of a book, then Seller sends a quote to both A and B. After that, A tries to negotiate a split of the payment with B, where A proposes how much A would like to pay. B only accepts the split when B pays no more than A pays, otherwise B rejects the split and the purchase is cancelled.

```
1  global protocol TwoBuyer(role A, role B, role S) {
2      bookId     (id: int) from A to S;
3      quoteA     (x: int)  from S to A; @"x >= 0"
4      quoteB     (y: int)  from S to B; @"x = y"
5      proposeA   (a: int)  from A to B; @"a >= 0 && a <= x"
6      choice at B {
7          ok     (b: int)  from B to A; @"b = y - a && b <= a"
8          buy    ()        from A to S;
9      } or {
10         no     ()        from B to A;
11         cancel ()        from A to S;
12     }
13 }
```

Figure 6.7: Two Buyer Protocol in Scribble

It is important to notice that refinements are projected to local roles in a way such that only variables known to that local role are projected. In this example, the Seller knows that they give the buyer A a quote x which is non-negative and they give the buyer B a quote y that is equal to x. However, A does know that the quote x is non-negative, but the same information is not perceived by B. This is because B is unaware of the

variable x in their local projection, and hence the refinement $x = y$ cannot be projected to B. Whereas it is obvious from the global perspective that y is also non-negative, such information is unfortunately not known to B.

It would be possible to address the issue via the use of existential quantifiers. While B does not learn the value of variable x, they know that there exists a variable x that the Seller sends to A. B could further learn that there exists an non-negative integer x and the received integer y is equal to that x, hence deduce that y is also non-negative. Unfortunately, neither Scribble nor FLUIDTYPES currently supports quantifiers.

We include the implementation for Seller and A in Appendix D.1, and focus on the implementation of B here, as shown in Figure 6.8. Although the partner of communication is not indicated clearly in each handler, the execution function is aware of the partner and ensures messages are sent to or received from the correct role.

```
1  module ImplB
2
3  open ScribbleGeneratedTwoBuyerB
4
5  let handlers = {
6      state30Onreceivequote B
7          = fun _ _ → ()
8      state32Onreceivepropose A
9          = fun _ _ → ()
10     state33
11         = fun st →
12             if st.y - st.a < st.a    // y - a is what B needs to pay
13             then State33Choice.ok
14             else State33Choice.no
15     state33Onsendok
16         = fun st →
17             if st.y - st.a < st.a    // This repeated check will
18             then st.y - st.a         // always pass
19             else failwith "Impossible"
20     state33Onsendno
21         = fun _ → ()
22 }
```

Figure 6.8: Implementation for Role B of Two Buyers Protocol

The choice at B (line 6 of Figure 6.7) corresponds to `state33` handler of label selection. We establish in line 12 of Figure 6.8 that if $y - a < a$, i.e. the `ok` branch is chosen when B pays less than A does, otherwise `no`. Since $b = y - a$ and $b \leq a$ both involve the payload variable b, there is no path constraint attached to this choice. Unfortunately, it is necessary to repeat the condition check in line 17 to satisfy the type system. It is not desirable to repeat the same check twice, since the second check will always pass and gives no extra information.

This limitation arises since each handler is checked separately with respect to their refinement types annotated in the type definition. In this case, there are no path constraints attached to either `ok` or `no` message in the protocol description. While it is possible to deduce that the `ok` message requires $y - a \leq a$ as a path constraint, by substituting b with $y - a$ from the equality refinement, the projection of refinements is syntax directed, and does not handle implied path constraints. Moreover, FLUIDTYPES is not able to establish the control flow of handlers and hence accumulate predicates along the control flow. The implementation of `state33` establishes a predicate for entering `ok` branch, but the handler for `ok` message is not aware of that established predicate.

Nonetheless, this example demonstrates that our FLUIDSESSION can be applied to multiparty protocols, not limited to binary protocols.

6.3 Example Protocols with Unsupported Refinements

Since our work depends on Scribble to produce a CFSM representation, we inherit the limitations from the prior work. In this section, we explore in detail how refinements can be extended to recursion in multiparty protocols.

Consider an accumulator protocol involving a Client and a Server, as shown in Figure 6.9. This is a recursive protocol, where the Client can choose to provide a number or quit. The Server is expected to add up all the numbers sent from the Client and send the accumulated value to the Client.

```
1  global protocol Accum(role C, role S) {
2      rec LOOP {
3          choice at C {
4              NUM   (x: int) from C to S;
5              ACCUM (y: int) from S to C;
6              continue LOOP;
7          } or {
8              BYE () from C to S;
9              BYE () from S to C;
10         }
11     }
12 }
```

Figure 6.9: Accumulator Protocol in Scribble

We would like to refine the protocol, such that the payload variable y in `ACCUM` message is the sum of all values of x communicated in `NUM` message. Currently, there is no way to express the intended properties. Since it is not possible to refer to variables in the previous iterations of the protocol, we cannot specify the *accumulation* of value. We propose a refined protocol in Figure 6.10, in an imaginary extension of Scribble.

We use $y_0 = 0$ to provide an initial value for y, and use $y = y' + x$ to specify that y in current iteration must be equal to the sum of y in previous iteration and x in current

```
global protocol Accum(role C, role S) {
    rec LOOP {
        @"y_0 = 0"
        choice at C {
            NUM   (x: int) from C to S;
            ACCUM (y: int) from S to C; @"y = y' + x"
            continue LOOP;
        } or {
            BYE () from C to S;
            BYE () from S to C;
        }
    }
}
```

Figure 6.10: Refined Accumulator Protocol in Imaginary Scribble

iteration. In this way, we can refer to payload variables across iterations and specify invariants about the loop.

In the CFSM representation, we can use two variables for y, one for the current iteration and one for the previous iteration. When the recursion occurs, we set the variable for previous iteration y' to that for the current iteration y, and continue with execution. We also need to unroll the recursion for the initial iteration, where there is no previous iteration, and use the initial value y_0 for the initial iteration of y.

On another persepective, our style of API generation keeps a backlog of the communication in the protocol. This may be sufficient for some use cases, for example those we have shown in the previous examples, but is unfit for this protocol. It is usually the case that the Client has a list of integers to process, and the list does not occur in the backlog of communications.

We propose to extend the handlers by a user-defined environment type, where a develop can use a type of their choice as the *environment* of the communication protocol. This is analogue to the concept of a state monad, but we choose not to use the name "state" for confusion avoidance. So the handlers take additional argument for environment, and an environment is part of the return type. We can use a polymorphic type `'env` for the type of environment:

```
state1OnsendNUM : 'env * State1 → int * 'env
state2OnreceiveACCUM : 'env * State2 → int → 'env
```

This proposal requires further improvement in λ^H and FLUIDTYPES, with support for polymorphism and recursive types. In addition, FLUIDTYPES currently does not have a model for built-in functions or data structures in F#, making it impossible to implement the protocol, even without refinements.

Chapter 7

Conclusion

7.1 Contributions

In this project, we develop and present an end-to-end framework for static verification of multiparty communication protocols with refinements. We begin with the protocol description language Scribble, extended with refinements [51, 36], and use the Scribble toolchain to obtain the projections of global protocols to local roles.

From that point on, we design and implement a code generation tool for F#, explained in Chapter 5, which uses the output from Scribble and produces handler-styled APIs for local roles. These APIs carry refinement types which can be validated by a typechecker extension we develop, and the handler style of API generation can guarantee the linear usage of channels.

We formalise a theory of refinement types in Chapter 3 and prove type safety and erasure properties. We implement it as a typechecker extension for F# and explain the design and implementation details in Chapter 4. The typechecker extension is able to verify the implementation of the protocol, with respect to refinements in the protocol, hence completing the end-to-end static verification of protocols.

Compared to previous work, we are not only able to avoid dynamic checks for refinements, but also linearity checks of communication channels during execution. We evaluate our framework via case studies in Chapter 6 and demonstrate the strengths and limitations with examples.

This project has received positive feedback from the community. We presented the work as *Fluid Types: Statically Verified Distributed Protocols with Refinements* at 11th Workshop on Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES 2019) [52], and in *Type My Morning* seminar series at Facebook London, and the community responded with positive feedback. We also had discussions with Don Syme, key designer and implementer of F#, about the work and received positive reviews from him.

Our F# refinement type extension is available in a public repository[1] under the MIT license. This library is not limited to checking multiparty protocol implementations but applies also to generic F# programs. F# users can use our refinement typechecker

independently from our work on code generation for session types. Similarly, the code generation tool for Scribble protocols is in a public repository[2] under the MIT license.

Overall, we offer a solid foundation to combine refinement types and multiparty session types, both in theory and in practice. Our static verification framework showcases the potential and the feasibility of the approach and invites further research into the intersection of two areas.

7.1.1 Open Source Contributions to F# Libraries

Our implementation makes use of several F# open source libraries. During the project period, we made the following contributions to open source libraries, including feature improvements and bug fixes:

- FsLexYacc

  Add `module` and `internal` option to FsLex (`https://github.com/fsprojects/FsLexYacc/pull/90`)

- FAKE

  Fix `Shell.mv` function (`https://github.com/fsharp/FAKE/pull/2309`)

- FSharp.TypeProviders.SDK

  Fix a crash due to `ToString` not overridden (`https://github.com/fsprojects/FSharp.TypeProviders.SDK/pull/310`)

- DotParser

  Handle escaped double quotes in parsing (`https://github.com/auduchinok/DotParser/pull/6`)

7.2 Future Work

This project opens path to future work on combining refinement types and session types. The following ideas are interesting open research questions that can be explored for an extension to this project.

7.2.1 Refinements in Multiparty Session Types

Rumyana et al. [36] propose a framework for specifying refinements in the context of multiparty session types. This project builds upon this setup, where refinements are attached as an assertion to the messages. The work focuses on practical aspects of implementation, but lacks a theoretical foundation of refinements in the MPST setup.

[1] `https://github.com/fangyi-zhou/FluidTypes`
[2] `https://github.com/fangyi-zhou/ScribbleCodeGen`

This work provides a formalisation for the refinement calculus λ^H in an isolated setup, without modelling the communication aspects. Therefore, the full meta-theory of MPST with refinement is not covered by this work and remains as future work. The meta-theory can provide the theoretical foundations for the safety claims in prior work and this work. Related work by Toninho et al. [46] formalises dependent session types with intuitionistic linear type theory, in the setting of π-calculus. Future work on the meta-theory with refinement types can compare its expressiveness with existing work.

Since this work uses a variant of λ-calculus with subtyping for the formalisation of λ^H, the meta-theory of MPST with refinements can be formalised in a variant of MPST with subtyping. In terms of expressiveness, the previous work is unable to express refinements in loop constructs in the protocol, such as invariants, or constraints on variables communicated in the previous iterations. The future work on the meta-theory may choose to include these extensions.

7.2.2 λ^H and FLUIDTYPES Library

Vazou et al. [48, 49] present the theory and implementation of LIQUIDHASKELL, a refinement type library for Haskell. Our work on FLUIDTYPES is heavily influenced by the work on LIQUIDHASKELL, but only implements a subset of the features available in refinement types in LIQUIDHASKELL.

Our implementation currently supports a subset of the features available in the F# language. In particular, many features frequently used by functional programmers are not supported, such as polymorphism, union types, tuples, etc. To improve usability, FLUIDTYPES needs to support more functionality available in F#.

Moreover, FLUIDTYPES lacks a model for built-in libraries for F#. While developers can use FLUIDTYPES for their own libraries and programs, only a small set of built-in libraries are modelled, for example, arithmetic and logical operators, integer literals and boolean literals. For a more usable refinement type library, it is necessary to provide models for built-in libraries, especially for data structures such as lists.

For the usability perspective, the error message quality needs improvement. Currently, FLUIDTYPES reports errors using the internal representation of λ^H instead of the surface language F#. This can confuse developers when an error occurs. Improvements in error reporting would improve the usability of the library.

Bibliography

[1] ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: a tutorial. *Computer 29*, 12 (Dec 1996), 66–76.

[2] BARRETT, C., FONTAINE, P., AND TINELLI, C. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016. Accessed on 20th May 2019.

[3] BENGTSON, J., BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND MAFFEIS, S. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS) 33*, 2 (2011), 8.

[4] BHARGAVAN, K., BOND, B., DELIGNAT-LAVAUD, A., FOURNET, C., HAWBLITZEL, C., HRITCU, C., ISHTIAQ, S., KOHLWEISS, M., LEINO, R., LORCH, J., MAILLARD, K., PANG, J., PARNO, B., PROTZENKO, J., RAMANANANDRO, T., RANE, A., RASTOGI, A., SWAMY, N., THOMPSON, L., WANG, P., ZANELLA-BÉGUELIN, S., AND ZINZINDOHOUÉ, J.-K. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages* (May 2017).

[5] BHARGAVAN, K., CORIN, R., DENIÉLOU, P., FOURNET, C., AND LEIFER, J. J. Cryptographic protocol synthesis and verification for multiparty sessions. In *2009 22nd IEEE Computer Security Foundations Symposium* (July 2009), pp. 124–140.

[6] BIERMAN, G. M., GORDON, A. D., HRIŢCU, C., AND LANGWORTHY, D. Semantic subtyping with an SMT solver. *Journal of Functional Programming 22*, 1 (2012), 31–105.

[7] BOUDOL, G. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992.

[8] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[9] COPPO, M., DEZANI-CIANCAGLINI, M., PADOVANI, L., AND YOSHIDA, N. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming* (2015), vol. 9104 of *LNCS*, Springer, pp. 146–178.

[10] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[11] DENIÉLOU, P.-M., AND YOSHIDA, N. Dynamic multirole session types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 435–446.

[12] DOWEK, G. The undecidability of typability in the lambda-pi-calculus. In *Typed Lambda Calculi and Applications* (Berlin, Heidelberg, 1993), M. Bezem and J. F. Groote, Eds., Springer Berlin Heidelberg, pp. 139–145.

[13] F# COMPILER SERVICE DEVELOPERS. F# Compiler Services. `http://fsharp.github.io/FSharp.Compiler.Service/`. Accessed on 21st May 2019.

[14] F# LANGUAGE REFERENCE AUTHORS. Attributes. `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/attributes`. Accessed on 21st May 2019.

[15] F# LANGUAGE REFERENCE AUTHORS. Code Quotations. `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/code-quotations`. Accessed on 21st May 2019.

[16] FOWLER, S. An Erlang Implementation of Multiparty Session Actors. In *ICE* (2016), vol. 223 of *EPTCS*, pp. 36–50.

[17] FREEMAN, T., AND PFENNING, F. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1991), PLDI '91, ACM, pp. 268–277.

[18] GAY, S. J., VASCONCELOS, V. T., RAVARA, A., GESBERT, N., AND CALDEIRA, A. Z. Modular Session Types for Distributed Object-oriented Programming. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2010), POPL '10, ACM, pp. 299–312.

[19] GIUNTI, M., AND VASCONCELOS, V. T. Linearity, session types and the pi calculus. *Mathematical Structures in Computer Science 26*, 2 (2016), 206–237.

[20] GORDON, A. F7: Refinement Types for F#. `https://www.microsoft.com/en-us/research/project/f7-refinement-types-for-f/`. Accessed on 23rd December 2018.

[21] HONDA, K., AND TOKORO, M. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming* (London, UK, UK, 1991), ECOOP '91, Springer-Verlag, pp. 133–147.

[22] HONDA, K., VASCONCELOS, V. T., AND KUBO, M. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems* (Berlin, Heidelberg, 1998), C. Hankin, Ed., Springer Berlin Heidelberg, pp. 122–138.

[23] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2008), POPL '08, ACM, pp. 273–284.

[24] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty Asynchronous Session Types. *Journal of the ACM 63* (2016), 1–67.

[25] HU, R. Distributed Programming Using Java APIs Generated from Session Types. *Behavioural Types: from Theory to Tools* (2017), 287–308.

[26] HU, R., AND YOSHIDA, N. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering* (2016), vol. 9633 of *LNCS*, Springer, pp. 401–418.

[27] HU, R., YOSHIDA, N., AND HONDA, K. Session-Based Distributed Programming in Java. In *22nd European Conference on Object-Oriented Programming* (2008), vol. 5142 of *LNCS*, Springer, pp. 516–541.

[28] IMAI, K., YOSHIDA, N., AND YUEN, S. Session-Ocaml: a Session-based Library with Polarities and Lenses. *Science of Computer Programming* (2018), 1–50.

[29] KNOWLES, K., AND FLANAGAN, C. Hybrid type checking. *ACM Trans. Program. Lang. Syst. 32*, 2 (Feb. 2010), 6:1–6:34.

[30] MARTIN-LÖF, P., AND SAMBIN, G. *Intuitionistic type theory*, vol. 9. Bibliopolis Naples, 1984.

[31] MICROSOFT RESEARCH, AND INRIA. F*: A Higher-Order Effectful Language Designed for Program Verification. `https://www.fstar-lang.org/`. Accessed on 9th January 2019.

[32] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348 – 375.

[33] MILNER, R. Functions as processes. *Mathematical Structures in Computer Science 2*, 2 (1992), 119–141.

[34] MILNER, R. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[35] NEYKOVA, R. Session Types Go Dynamic or How to Verify Your Python Conversations. In *5th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software* (2013), vol. 137 of *EPTCS*, Open Publishing Association, pp. 95–102.

[36] NEYKOVA, R., HU, R., YOSHIDA, N., AND ABDELJALLAL, F. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. In *27th International Conference on Compiler Construction* (2018), ACM, pp. 128–138.

[37] NEYKOVA, R., YOSHIDA, N., AND HU, R. SPY: Local Verification of Global Protocols. In *4th International Conference on Runtime Verification* (2013), vol. 8174 of *LNCS*, Springer, pp. 363–358.

[38] NG, N., COUTINHO, J. G., AND YOSHIDA, N. Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction* (2015), vol. 9031 of *LNCS*, Springer, pp. 212–232.

[39] NG, N., YOSHIDA, N., AND HONDA, K. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *50th International Conference on Objects, Models, Components, Patterns* (2012), vol. 7304 of *LNCS*, Springer, pp. 202–218.

[40] PETRICEK, T., GUERRA, G., AND SYME, D. Types from data: Making structured data first-class citizens in F#. In *Proceedings of Conference on Programming Language Design and Implementation* (2016), PLDI 2016.

[41] PIERCE, B. C., AND TURNER, D. N. Local type inference. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California* (1998). Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

[42] PROJECT EVEREST. Project Everest. `https://project-everest.github.io`. Accessed on 9th January 2019.

[43] RONDON, P. M., KAWAGUCI, M., AND JHALA, R. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 159–169.

[44] SCHMID, G. S., AND KUNCAK, V. SMT-based checking of predicate-qualified types for Scala. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala* (2016), ACM, pp. 31–40.

[45] SWAMY, N., CHEN, J., FOURNET, C., STRUB, P.-Y., BHARGAVAN, K., AND YANG, J. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2011), ICFP '11, ACM, pp. 266–278.

[46] TONINHO, B., CAIRES, L., AND PFENNING, F. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming* (New York, NY, USA, 2011), PPDP '11, ACM, pp. 161–172.

[47] VASCONCELOS, V. T. Fundamentals of session types. *Information and Computation 217* (2012), 52 – 70.

[48] VAZOU, N., SEIDEL, E. L., JHALA, R., VYTINIOTIS, D., AND PEYTON-JONES, S. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2014), ICFP '14, ACM, pp. 269–282.

[49] VAZOU, N., TONDWALKAR, A., CHOUDHURY, V., SCOTT, R. G., NEWTON, R. R., WADLER, P., AND JHALA, R. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang. 2*, POPL (Dec. 2017), 53:1–53:31.

[50] YOSHIDA, N. Lecture notes for CO406H: Concurrent Processes. `https://www.doc.ic.ac.uk/~yoshida/slides/lectures/NY_PI_15_16.pdf`. Accessed on 12th January 2019.

[51] YOSHIDA, N., HU, R., NEYKOVA, R., AND NG, N. The scribble protocol language. In *International Symposium on Trustworthy Global Computing* (2013), Springer, pp. 22–41.

[52] ZHOU, F., FERREIRA, F., NEYKOVA, R., AND YOSHIDA, N. Fluid Types: Statically Verified Distributed Protocols with Refinements. PLACES 2019 11th Workshop on Programming Language Approaches to Concurrency- & Communication-cEntric Software, 2019.

Appendix A

A Basic Type System for the Simply Typed λ-calculus

A.1 Types and Typing Judgements

We define syntax of types in λ-calculus in Figure A.1.

$$
\begin{array}{rr}
t ::= & \text{Types} \\
\quad b & \text{Base Type (See Figure 3.1)} \\
\quad \mid t \to t & \text{Function Type}
\end{array}
$$

Figure A.1: Types in λ-calculus

We present the typing judgements for λ-calculus in Figure A.2.

$$\frac{}{\Gamma, x : t, \Gamma' \vdash x : t}\ \text{T-VAR}$$

$$\frac{}{\Gamma \vdash c : \mathsf{Ty'}(c)}\ \text{T-CONST}$$

$$\frac{\Gamma \vdash M_1 : t_1 \to t_2 \qquad \Gamma \vdash M_2 : t_1}{\Gamma \vdash M_1 M_2 : t_2}\ \text{T-APP}$$

$$\frac{\Gamma, x : t_1; \Delta \vdash M : t_2}{\Gamma \vdash \lambda x.M : t_1 \to t_2}\ \text{T-ABS}$$

$$\frac{\Gamma \vdash M_1 : \texttt{bool} \qquad \Gamma \vdash M_2 : t \qquad \Gamma \vdash M_3 : t}{\Gamma \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 : t}\ \text{T-COND}$$

Figure A.2: Typing Judgements for λ-calculus

Types for some constants $\mathsf{Ty'}(c)$ are shown in Table A.1.

1	::	$\texttt{int}$	integers
$\texttt{true}$	::	$\texttt{bool}$	booleans
$(+)$	::	$\texttt{int} \to \texttt{int} \to \texttt{int}$	arithmetic
$(=)_{\texttt{int}}$	::	$\texttt{int} \to \texttt{int} \to \texttt{bool}$	relational

Table A.1: Constants and their Basic Types

Appendix B

Lemmas and Proofs

B.1 Lemmas and Proofs for Chapter 3

Lemma B.1.1. *If* $\Gamma; \Delta \vdash \tau <: \sigma$, *then* $\text{erase}(\tau) = \text{erase}(\sigma)$.

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \tau <: \sigma$.

1. $\Gamma; \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}$ (SUB-BASE)
   Since $\text{erase}(\{\nu : b \mid M_1\}) = b$ and $\text{erase}(\{\nu : b \mid M_2\}) = b$, the property holds.
2. $\Gamma; \Delta \vdash ((x : \tau_1) \to \tau_2) <: ((x : \sigma_1) \to \sigma_2)$ (SUB-FUNC)
   From (SUB-FUNC), we have $\Gamma; \Delta \vdash \sigma_1 <: \tau_1$, and $\Gamma, x : \sigma_1; \Delta \vdash \tau_2 <: \sigma_2$.
   By inductive hypothesis, we have $\text{erase}(\sigma_1) = \text{erase}(\tau_1)$ and $\text{erase}(\tau_2) = \text{erase}(\sigma_2)$.
   Since $\text{erase}((x : \sigma_1) \to \sigma_2) = \text{erase}(\sigma_1) \to \text{erase}(\sigma_2)$ and $\text{erase}((x : \tau_1) \to \tau_2) = \text{erase}(\tau_2) \to \text{erase}(\tau_2)$, by substitution, the property holds.

□

Lemma B.1.2. *If* $\Gamma; \Delta \vdash \tau$, *then* $\text{fv}(\tau) \subseteq \text{dom}(\Gamma)$

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \tau$.

1. $\Gamma; \Delta \vdash \{\nu : b \mid M\}$ (WF-TY-BASE)
   By (WF-TY-BASE), $\text{fv}(\{\nu : b \mid M\}) \subseteq \text{dom}(\Gamma)$.
2. $\Gamma; \Delta \vdash ((x : \tau_1) \to \tau_2)$ (WF-TY-FUNC)
   By (WF-TY-BASE), $\text{fv}((x : \tau_1) \to \tau_2) \subseteq \text{dom}(\Gamma)$.

□

Lemma B.1.3. *If* $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \rho$ *and* $x \notin \text{fv}(\sigma)$, *then* $\Gamma, y : \sigma, x : \tau, \Gamma'; \Delta \vdash \rho$

Proof. By induction on the derivation of $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \rho$.

1. $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \{\nu : b \mid M\}$ (WF-TY-BASE)

   From (WF-TY-BASE), we have $\Gamma, x : \tau, y : \sigma, \Gamma', \nu : b; \Delta \vdash_{\text{erase}} M : \texttt{bool}$, and $\text{fv}(\{\nu : b \mid M\}) \subseteq \text{dom}(\Gamma, x : \tau, y : \sigma, \Gamma')$.

   By permutation lemma of simple λ-calculus, we have $\Gamma, y : \sigma, x : \tau, \Gamma', \nu : b; \Delta \vdash_{\text{erase}} M : \texttt{bool}$.

   Note that $\text{dom}(\Gamma, x : \tau, y : \sigma, \Gamma') = \text{dom}(\Gamma) \cup \{x, y\} \cup \text{dom}(\Gamma') = \text{dom}(\Gamma, y : \sigma, x : \tau, \Gamma')$

   By (WF-TY-BASE), we obtain $\Gamma, y : \sigma, x : \tau, \Gamma'; \Delta \vdash \{\nu : b \mid M\}$.

2. $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash ((z : \rho_1) \to \rho_2)$

   From (WF-TY-FUNC), we have $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \rho_1$, $\Gamma, x : \tau, y : \sigma, \Gamma', z : \rho_1; \Delta \vdash \rho_2$, and $\text{fv}((z : \rho_1) \to \rho_2) \subseteq \text{dom}(\Gamma, x : \tau, y : \sigma, \Gamma')$.

   By inductive hypothesis, we have $\Gamma, y : \sigma, x : \tau, \Gamma; \Delta \vdash \rho_1$ and $\Gamma, y : \sigma, x : \tau, \Gamma', z : \rho_1; \Delta \vdash \rho_2$ (applying inductive hypothesis on $\Gamma', z : \rho_1$).

   Note that $\text{dom}(\Gamma, x : \tau, y : \sigma, \Gamma') = \text{dom}(\Gamma) \cup \{x, y\} \cup \text{dom}(\Gamma') = \text{dom}(\Gamma, y : \sigma, x : \tau, \Gamma')$

   By (WF-TY-FUNC), we obtain $\Gamma, y : \sigma, x : \tau, \Gamma'; \Delta \vdash ((z : \rho_1) \to \rho_2)$.

□

Lemma B.1.4. *If* $y \notin \text{dom}(\Gamma)$, *and* $\Gamma; \Delta \vdash \sigma$, *and* $\Gamma; \Delta \vdash \tau$, *then* $\Gamma, y : \sigma; \Delta \vdash \tau$

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \tau$.

1. $\Gamma; \Delta \vdash \{\nu : b \mid M\}$ (WF-TY-BASE)

   From (WF-TY-BASE), we have $\Gamma, \nu : b; \Delta \vdash_{\text{erase}} M : \texttt{bool}$, and $\text{fv}(\{\nu : b \mid M\}) \subseteq \text{dom}(\Gamma)$.

   By weakening and permutation lemma of simple λ-calculus, $\Gamma, y : \sigma, \nu : b; \Delta \vdash_{\text{erase}} M : \texttt{bool}$.

   Note that $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma, y : \sigma) = \text{dom}(\Gamma) \cup \{y\}$, hence we have $\text{fv}(\{\nu : b \mid M\}) \subseteq \text{dom}(\Gamma, y : \sigma)$.

   By (WF-TY-BASE), $\Gamma, y : \sigma; \Delta \vdash \{\nu : b \mid M\}$.

2. $\Gamma; \Delta \vdash ((x : \tau_1) \to \tau_2)$ (WF-TY-FUNC)

   From (WF-TY-FUNC), we have $\Gamma; \Delta \vdash \tau_1$, $\Gamma, x : \tau_1; \Delta \vdash \tau_2$ and $\text{fv}((x : \tau_1) \to \tau_2) \subseteq \text{dom}(\Gamma)$.

   By inductive hypothesis, we have $\Gamma, y : \sigma; \Delta \vdash \tau_1$ and $\Gamma, y : \sigma, x : \tau_1; \Delta \vdash \tau_2$

   Since $y \notin \text{dom}(\Gamma)$ and $\text{fv}((x : \tau_1) \to \tau_2) \subseteq \text{dom}(\Gamma)$ (By Lemma B.1.2), we have $y \notin \text{fv}((x : \tau_1) \to \tau_2)$.

   By Lemma B.1.3, we have $\Gamma, x : \tau_1, y : \sigma; \Delta \vdash \tau_2$

   Note that $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma, y : \sigma) = \text{dom}(\Gamma) \cup \{y\}$, hence we have $\text{fv}((x : \tau_1) \to \tau_2) \subseteq \text{dom}(\Gamma, y : \sigma)$.

   By (WF-TY-FUNC), we have $\Gamma, y : \sigma; \Delta \vdash \tau_2$.

□

Lemma B.1.5. *If* $\Gamma; \Delta \vdash \sigma$ *and* $y \notin \mathrm{dom}(\Gamma)$, *then* $\mathsf{Valid}([\![\Gamma, y : \sigma]\!] \implies [\![\Gamma]\!])$

Proof. By induction on the structure of σ.

1. $\{\nu : b \mid M\}$
   $[\![\Gamma, y : \{\nu : b \mid M\}]\!] = [\![\Gamma]\!] \wedge [\![M[y/\nu]]\!]$
   By $\wedge E_1$ on the right hand side, we have $[\![\Gamma]\!]$.
2. $(x : \tau_1) \to \tau_2$
   $[\![\Gamma, y : (x : \tau_1) \to \tau_2]\!] = [\![\Gamma]\!]$.
   Trivial.

□

Lemma B.1.6. *If* $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \rho_1 <: \rho_2$ *and* $x \notin \mathrm{fv}(\sigma)$, *then* $\Gamma, y : \sigma, x : \tau, \Gamma'; \Delta \vdash \rho_1 <: \rho_2$

Proof. By induction on the derivation of $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \rho_1 <: \rho_2$.

1. $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}$ (SUB-BASE)
   From (SUB-BASE), we have $\mathsf{Valid}([\![\Gamma, x : \tau, y : \sigma, \Gamma']\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])$.
   Notice that $[\![\Gamma, x : \tau, y : \sigma, \Gamma']\!] \iff [\![\Gamma, y : \sigma, x : \tau, \Gamma']\!]$, this is due to the commutativity of $\wedge$ and follows from Definition 3.4.1.
   We have $\mathsf{Valid}([\![\Gamma, y : \sigma, x : \tau, \Gamma']\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])$. By (SUB-BASE), we have $\Gamma, y : \sigma, x : \tau, \Gamma' \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}$.
2. $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash ((z : \rho_1) \to \rho_2) <: ((z : \omega_1) \to \omega_2)$ (SUB-FUNC)
   From (SUB-FUNC), we have $\Gamma, x : \tau, y : \sigma, \Gamma'; \Delta \vdash \omega_1 <: \rho_1$, and $\Gamma, x : \tau, y : \sigma, \Gamma', z : \rho_1; \Delta \vdash \omega_2 <: \rho_2$.
   By inductive hypothesis, we have $\gamma, y : \sigma, x : \tau, \Gamma'; \Delta \vdash \omega_1 <: \rho_1$, and $\Gamma, y : \sigma, x : \tau, \Gamma', z : \rho_1; \Delta \vdash \omega_2 <: \rho_2$.
   By (SUB-FUNC), we have $\Gamma, y : \sigma, x : \tau, \Gamma'; \Delta \vdash ((z : \rho_1) \to \rho_2) <: ((z : \omega_1) \to \omega_2)$.

□

Lemma B.1.7. *If* $y \notin \mathrm{dom}(\Gamma)$, *and* $\Gamma; \Delta \vdash \sigma$, *and* $\Gamma; \Delta \vdash \tau_1 <: \tau_2$, *then* $\Gamma, y : \sigma; \Delta \vdash \tau_1 <: \tau_2$

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \tau_1 <: \tau_2$.

1. $\Gamma; \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}$ (SUB-BASE)
   From (SUB-BASE), we have $\mathsf{Valid}([\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])$.
   By Lemma B.1.5, we have $\mathsf{Valid}([\![\Gamma, y : \sigma]\!] \implies [\![\Gamma]\!])$.
   We prove the validity of $[\![\Gamma, y : \sigma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$ by natural deduction.

1.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$	premise
2.	$[\![\Gamma, y : \sigma]\!] \implies [\![\Gamma]\!]$	Lemma B.1.5
3.	$[\![\Gamma, y : \sigma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!]$	assumption
4.	$[\![\Gamma, y : \sigma]\!]$	$\wedge E_1(3)$
5.	$[\![\Delta]\!] \wedge [\![M_1]\!]$	$\wedge E_2(3)$
6.	$[\![\Gamma]\!]$	$\implies E(2,4)$
7.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!]$	$\wedge I(6,5)$
8.	$[\![M_2]\!]$	$\implies E(1,7)$
9.	$[\![\Gamma, y : \sigma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$	$\implies I(3,8)$

Hence we have $\mathsf{Valid}([\![\Gamma, y : \sigma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])$.

By (SUB-BASE), we have $\Gamma, y : \sigma; \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}$.

2. $\Gamma; \Delta \vdash ((x : \tau_1) \to \tau_2) <: ((x : \rho_1) \to \rho_2)$ (SUB-FUNC)

   From (SUB-FUNC), we have $\Gamma; \Delta \vdash \rho_1 <: \tau_1$, and $\Gamma, x : \rho_1; \Delta \vdash \tau_2 <: \rho_2$.

   By inductive hypothesis, we have $\Gamma, y : \sigma; \Delta \vdash \rho_1 <: \tau_1$ and $\Gamma, x : \rho_1, y : \sigma; \Delta \vdash \tau_2 <: \rho_2$.

   By Lemma B.1.2, we have $\mathrm{fv}(\sigma) \subseteq \mathrm{dom}(\Gamma)$. Since $y \notin \mathrm{dom}(\Gamma)$, we have $y \notin \mathrm{fv}(\sigma)$.

   By Lemma B.1.6, we have $\Gamma, y : \sigma, x : \rho_1; \Delta; \vdash \tau_2 <: \rho_2$.

   By (SUB-FUNC), we have $\Gamma, y : \sigma; \Delta \vdash ((x : \tau_1) \to \tau_2) <: ((x : \rho_1) \to \rho_2)$.

□

Lemma B.1.8. *If $x \neq y$, and $\vdash \Gamma, x : \tau, \Gamma'; \Delta$, and $\Gamma, \Gamma'; \Delta \vdash M \Rightarrow \tau$, and $\Gamma, x : \tau, \Gamma'; \Delta \vdash y \Rightarrow \sigma$, then $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash y \Rightarrow \sigma[M/x]$.*

Proof. $\Gamma, x : \tau, \Gamma'; \Delta \vdash y \Rightarrow \sigma$ has two possible derivations, via (TY-VAR-BASE) or (TY-VAR-FUNC). We consider by case here.

Since x cannot occur twice in the typing context, we have $x \notin \mathrm{dom}(\Gamma)$ and $x \notin \mathrm{dom}(\Gamma')$. From the derivation, we know that $y \in \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')$.

- (TY-VAR-BASE)

  We have $\Gamma, x : \tau, \Gamma'; \Delta \vdash y \Rightarrow \{\nu : b \mid \nu = y\}$, and that the context $\Gamma, x : \tau, \Gamma'$ can be split into $\Gamma_1, y : \{\nu : b \mid M\}, \Gamma_2$ for some base type b and term M.

  In either case $x \in \mathrm{dom}(\Gamma_1)$ or $x \in \mathrm{dom}(\Gamma_2)$, we have $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash y \Rightarrow \{\nu : b \mid \nu = y\}$ by (TY-VAR-BASE), and that $(\nu = y)[M/x] = \nu = y$, as required.

- (TY-VAR-FUNC)

  We have $\Gamma, x : \tau, \Gamma'; \Delta \vdash y \Rightarrow (z : \rho_1) \to \rho_2$ and $\sigma = (z : \rho_1) \to \rho_2$.

– Suppose $y \in \mathrm{dom}(\Gamma)$, there must exist a split of $\Gamma = \Gamma_1, y : \sigma, \Gamma_2$. Hence we have $\vdash \Gamma_1, y : \sigma, \Gamma_2, x : \tau, \Gamma'; \Delta$;

From Lemma B.1.2, we have $\mathrm{fv}(\sigma) \subseteq \mathrm{dom}(\Gamma_1)$. Since $\Gamma = \Gamma_1, y : \sigma, \Gamma_2$, we have $\mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma)$.

Since $x \notin \mathrm{dom}(\Gamma)$, we also have $x \notin \mathrm{fv}(\sigma)$, therefore $\sigma[M/x] = \sigma$.

By (TY-VAR-FUNC), we have $\Gamma_1, y : \sigma, \Gamma_2, \Gamma'[M/x]; \Delta[M/x] \vdash y \Rightarrow \sigma[M/x]$, where $\Gamma = \Gamma_1, y : \sigma, \Gamma_2$ and $\sigma[M/x] = \sigma$.

– Suppose $y \in \mathrm{dom}(\Gamma')$, there must exist a split of $\Gamma' = \Gamma_1, y : \sigma, \Gamma_2$. Therefore, $\Gamma'[M/x] = \Gamma_1[M/x], y : \sigma[M/x], \Gamma_2[M/x]$.

By (TY-VAR-FUNC), we have $\Gamma, \Gamma_1[M/x], y : \sigma[M/x], \Gamma_2[M/x]; \Delta[M/x] \vdash y \Rightarrow \sigma[M/x]$, where $\Gamma'[M/x] = \Gamma_1[M/x], y : \sigma[M/x], \Gamma_2[M/x]$.

□

Lemma B.1.9. *If* $\Gamma, x : \tau, \Gamma'; \Delta \vdash \sigma$ *and* $\Gamma, x : \tau, \Gamma'; \Delta \vdash M \Rightarrow \tau$, *then* $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash \sigma[M/x]$

Proof. By induction on the derivation of $\Gamma, x : \tau, \Gamma'; \Delta \vdash \sigma$. □

Lemma B.1.10. *The following properties hold:*

1. $[\![M[N/y]]\!] = [\![M]\!][[\![N]\!]/y]$.
2. $[\![\Delta[N/y]]\!] = [\![\Delta]\!][[\![N]\!]/y]$.
3. $[\![\Gamma[N/y]]\!] = [\![\Gamma]\!][[\![N]\!]/y]$.

Proof. By induction on Definition 3.4.1. □

Lemma B.1.11. *If* $\Gamma, x : \tau, \Gamma'; \Delta \vdash \sigma_1 <: \sigma_2$ *and* $\Gamma, x : \tau, \Gamma'; \Delta \vdash M \Rightarrow \tau$, *then* $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash \sigma_1[M/x] <: \sigma_2[M/x]$.

Proof. By induction on the derivation of $\Gamma, x : \tau, \Gamma'; \Delta \vdash \sigma_1 <: \sigma_2$.

1. $\Gamma, x : \tau, \Gamma'; \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}$ (SUB-BASE)

   From (SUB-BASE), we have $\mathsf{Valid}([\![\Gamma, x : \tau, \Gamma']\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])$. Denote this logical formula as A.

   We first show $\mathsf{Valid}([\![\Gamma, x : \tau, \Gamma']\!] \implies [\![\Gamma, \Gamma']\!])$ (denoted as B) is valid:

   By Definition 3.4.1

$$[\![\Gamma, x : \tau, \Gamma']\!] = \begin{cases} [\![\Gamma]\!] \wedge [\![\Gamma']\!] & \text{if } \tau \text{ is } (x : \tau_1) \to \tau_2 \\ [\![\Gamma]\!] \wedge [\![M'[x/\nu]]\!] \wedge [\![\Gamma']\!] & \text{if } \tau \text{ is } \{\nu : b \mid M'\} \end{cases}$$

   From A and B, we have $\mathsf{Valid}([\![\Gamma, \Gamma']\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])$ (denoted as C).

We substitute all occurences of the variable x to $[\![M]\!]$ and obtain $\mathsf{Valid}((\![\Gamma, \Gamma']\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!])[M/x])$

By Lemma B.1.10, we have
$\mathsf{Valid}((\![\Gamma[M/x], \Gamma'[M/x]]\!] \wedge [\![\Delta[M/x]]\!] \wedge [\![M_1[M/x]]\!] \implies [\![M_2[M/x]]\!]))$

Since a variable cannot occur in a context twice, we have $x \notin \mathrm{dom}(\Gamma)$, and hence $\Gamma[M/x] = \Gamma$.

By (SUB-BASE), we have
$\Gamma, \Gamma[M/x]; \Delta[M/x] \vdash \{v : b \mid M_1[M/x]\} <: \{v : b \mid M_2[M/x]\}$.

2. $\Gamma, x : \tau, \Gamma'; \Delta \vdash ((y : \sigma_1) \to \sigma_2) <: ((y : \rho_1) \to \rho_2)$ (SUB-FUNC)

   By inductive hypothesis.

□

Lemma B.1.12. *Suppose* $\vdash \Gamma, x : \tau, \Gamma'; \Delta$ *and* $\Gamma, x : \tau, \Gamma'; \Delta \vdash x \Rightarrow \tau$.

1. *If* $\Gamma, \Gamma'; \Delta \vdash M \Rightarrow \tau$, *Then* $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash M \Rightarrow \tau[M/x]$.
2. *If* $\Gamma, \Gamma'; \Delta \vdash M \Leftarrow \tau$, *Then* $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash M \Leftarrow \tau[M/x]$.

Proof. By simultaneous induction on the derivation of $\Gamma, \Gamma'; \Delta \vdash M \Rightarrow \tau$ and $\Gamma, \Gamma'; \Delta \vdash M \Leftarrow \tau$.

1. $\Gamma, \Gamma'; \Delta \vdash y \Rightarrow \tau$ (TY-VAR-BASE), (TY-VAR-FUNC)

   By application of the same argument in Lemma B.1.8.

2. $\Gamma, \Gamma'; \Delta \vdash c \Rightarrow \tau$ (TY-CONST)

   Constant types have no free variables, hence the substitution $\tau[M/x] = \tau$. Moreover, there exists a derivation of constants for any typing context.

   By (TY-CONST), we have $\Gamma, \Gamma'[M/x]; \Delta \vdash c \Rightarrow \tau[M/x]$.

3. $\Gamma, \Gamma'; \Delta \vdash \lambda y.M \Leftarrow (y : \tau_1) \to \tau_2$ (TY-ABS)

   From (TY-ABS) we have $\Gamma, \Gamma', y : \tau_1; \Delta \vdash M \Leftarrow \tau_2$ and $\Gamma, \Gamma'; \Delta \vdash ((y : \tau_1) \to \tau_2)$.

   From inductive hypothesis, we have $\Gamma, \Gamma'[M/x], y : \tau_1[M/x]; \Delta[M/x] \vdash M \Leftarrow \tau_2[M/x]$.

   By Lemma B.1.9, we have $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash ((y : \tau_1) \to \tau_2)[M/x]$. Note that $((y : \tau_1) \to \tau_2)[M/x] = ((y : \tau_1[M/x]) \to \tau_2[M/x])$.

   We apply (TY-ABS) and obtain $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash M \Leftarrow ((y : \tau_1) \to \tau_2)[M/x]$, as required.

4. $\Gamma, \Gamma'; \Delta \vdash M_1 M_2 \Rightarrow \tau_2[M_2/y]$ (TY-APP)

   From (TY-APP) we have $\Gamma, \Gamma'; \Delta \vdash M_1 \Rightarrow ((y : \tau_1) \to \tau_2)$ and $\Gamma, \Gamma'; \Delta \vdash M_2 \Leftarrow \tau_1$ for some τ_1.

   From inductive hypothesis, we have $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash M_1 \Rightarrow ((y : \tau_1) \to \tau_2)[M/x]$ and $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash M_2 \Leftarrow \tau_1[M/x]$.

   Note that $((y : \tau_1) \to \tau_2)[M/x] = ((y : \tau_1[M/x]) \to \tau_2[M/x])$, hence we can apply both rules to (TY-APP) and obtain $\Gamma, \Gamma'[M/x]; \Delta[M/x] \vdash M_1 M_2 \Rightarrow \tau_2[M/x]$.

5. $\Gamma, \Gamma'; \Delta \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 \Leftarrow \tau$ (TY-COND)

   By inductive hypothesis and Lemma B.1.9.

6. $\Gamma, \Gamma'; \Delta \vdash (M : \tau) \Rightarrow \tau$ (TY-ANNO)

   By inductive hypothesis.

7. $\Gamma, \Gamma'; \Delta \vdash M \Leftarrow \tau'$ (TY-SUB)

   By inductive hypothesis, Lemma B.1.9 and Lemma B.1.11.

□

Lemma B.1.13 (Weakening (Typing Context))**.** *Suppose $y \notin \mathrm{dom}(\Gamma)$ and $\Gamma; \Delta \vdash \sigma$, then*

1. *If $\Gamma; \Delta \vdash M \Rightarrow \tau$, then $\Gamma, y : \sigma; \Delta \vdash M \Rightarrow \tau$.*
2. *If $\Gamma; \Delta \vdash M \Leftarrow \tau$, then $\Gamma, y : \sigma; \Delta \vdash M \Leftarrow \tau$.*

Proof. By simultaneous induction on $\Gamma; \Delta \vdash M \Rightarrow \tau$ and $\Gamma; \Delta \vdash M \Leftarrow \tau$ with Lemma B.1.4 and Lemma B.1.7. □

Lemma B.1.14. *If $[\![\Delta']\!] \implies [\![\Delta]\!]$ and $\Gamma; \Delta \vdash \tau$, then $\Gamma; \Delta' \vdash \tau$.*

Proof. We notice that the derivation of $\Gamma; \Delta \vdash \tau$ does not depend on Δ. Hence the results hold by induction on the derivation of $\Gamma; \Delta \vdash \tau$. □

Lemma B.1.15. *If $[\![\Delta']\!] \implies [\![\Delta]\!]$ and $\Gamma; \Delta \vdash \tau <: \sigma$, then $\Gamma; \Delta' \vdash \tau <: \sigma$.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \tau <: \sigma$.

1. $\Gamma; \Delta \vdash \{\nu : b \mid M_1\} <: \{\nu : b \mid M_2\}$ (SUB-BASE)

   From (SUB-BASE), we know that $[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$.

   We show that $[\![\Gamma]\!] \wedge [\![\Delta']\!] \wedge M_1 \implies [\![M_2]\!]$.

1.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$	premise
2.	$[\![\Delta']\!] \implies [\![\Delta]\!]$	premise
3.	$[\![\Gamma]\!] \wedge [\![\Delta']\!] \wedge [\![M_1]\!]$	assumption
4.	$[\![\Delta']\!]$	$\wedge E_2(3)$
5.	$[\![\Delta]\!]$	$\implies E(2,4)$
6.	$[\![\Gamma]\!]$	$\wedge E_1(3)$
7.	$[\![M_1]\!]$	$\wedge E_3(3)$
8.	$[\![\Gamma]\!] \wedge [\![\Delta]\!] \wedge [\![M_1]\!]$	$\wedge I(6,5,7)$
9.	$[\![M_2]\!]$	$\implies E(1,8)$
10.	$[\![\Gamma]\!] \wedge [\![\Delta']\!] \wedge [\![M_1]\!] \implies [\![M_2]\!]$	$\implies I(3,9)$

2. $\Gamma; \Delta \vdash ((x : \tau_1) \to \tau_2) <: ((x : \sigma_1) <: \sigma_2)$ (SUB-FUNC)

   By inductive hypothesis.

□

Lemma B.1.16 (Weakening (Predicate Context))**.** *Suppose* $[\![\Delta']\!] \implies [\![\Delta]\!]$*, then*

1. *If* $\Gamma; \Delta \vdash M \Rightarrow \tau$*, then* $\Gamma; \Delta' \vdash M \Rightarrow \tau$*.*
2. *If* $\Gamma; \Delta \vdash M \Leftarrow \tau$*, then* $\Gamma; \Delta' \vdash M \Leftarrow \tau$*.*

Proof. By mutual induction on the derivation of $\Gamma; \Delta \vdash M \Rightarrow \tau$ and $\Gamma; \Delta \vdash M \Leftarrow \tau$, with Lemma B.1.14 and Lemma B.1.15. □

Appendix C

Syntax of Refinement Annotations

C.1 Syntax of Refinement Type Annotation in F#

⟨*ident*⟩ ::= `[A-Za-z][A-Za-z0-9_]*`

⟨*int*⟩ ::= integers

⟨*bool*⟩ ::= `true` | `false`

⟨*arith*⟩ ::= `+` | `-`

⟨*logic*⟩ ::= `&&` | `||`

⟨*rel*⟩ ::= `=` | `>` | `<` | `>=` | `<=` | `<>`

⟨*expr0*⟩ ::= ⟨*expr0*⟩ ⟨*logic*⟩ ⟨*expr0*⟩ | ⟨*expr1*⟩

⟨*expr1*⟩ ::= `not` ⟨*expr1*⟩ | ⟨*expr1*⟩ ⟨*rel*⟩ ⟨*expr1*⟩ | ⟨*expr2*⟩

⟨*expr2*⟩ ::= ⟨*expr2*⟩ ⟨*arith*⟩ ⟨*expr2*⟩ | ⟨*expr3*⟩

⟨*expr3*⟩ ::= ⟨*expr3*⟩ `$` ⟨*ident*⟩ | ⟨*expr4*⟩

⟨*expr4*⟩ ::= ⟨*ident*⟩ | `(` ⟨*expr0*⟩ `)` | ⟨*int*⟩ | ⟨*bool*⟩

⟨*ty*⟩ ::= `(` ⟨*ident*⟩ `:` ⟨*ty*⟩ `)` `->` ⟨*ty*⟩ | `{` ⟨*ident*⟩ `:` ⟨*ident*⟩ `|` ⟨*expr0*⟩ `}` | ⟨*ident*⟩

Appendix D

Code for Evaluation

D.1 Implementation for Two Buyers

```
module ImplA

open ScribbleGeneratedTwoBuyerA

let randomInt () = System.Random().Next()

let handlers = {
    state19Onsendbook Id
        = fun _ → randomInt ()
    state21Onreceivequote A
        = fun _ _ → ()
    state22Onsendpropose A
        = fun st →
            if st.x >= 5 then st.x - 5 else 0
    state23Onreceiveok
        = fun _ _ → ()
    state23Onreceiveno
        = fun _ → ()
    state24Onsendcancel
        = fun _ → ()
    state25Onsendbuy
        = fun _ → ()
}
```

Figure D.1: Implementation for Role A of Two Buyers Protocol

```
module ImplS

open ScribbleGeneratedTwoBuyerS

let randomInt () = System.Random().Next()

let handlers = {
    state7Onreceivebookid
        = fun _ _ → ()
    state9Onsendquotea
        = fun _ →
            let rnd = randomInt ()
            if rnd >= 0 then rnd else -rnd
    state10Onsendquoteb
        = fun st → st.x
    state11Onreceivebuy
        = fun _ → ()
    state11Onreceivecancel
        = fun _ → ()
}
```

Figure D.2: Implementation for Role S of Two Buyers Protocol