UNIVERSITY OF SUSSEX

DOCTORAL THESIS

On Implicit Program Constructs

Author:
Alexander Paul JEFFERY

Supervisor:
Dr. Martin BERGER

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

in the group

Foundations of Software Systems
School of Engineering and Informatics

September 2019

Declaration of Authorship

I, Alexander Paul JEFFERY, declare that this thesis titled, "On Implicit Program Constructs" and the work presented in it are my own. I confirm that:

- This thesis has not been and will no be, submitted in whole or in part to another University for the award of any other degree.
- Sources of information have been clearly stated or referenced in the bibliography. Any content that is joint work with others is clearly stated as such and all contributors are named.

This thesis contains content adapted from papers authored by myself, and in some cases co-authored by my supervisor, Dr. Martin BERGER. All portions of this work adapted from other works are clearly indicated as such, and those portions of this work that are joint work are clearly indicated in the headings of the chapters in which they appear, and the degree of collaboration is indicated.

Signed:

Date:

"Blessed is the one who finds wisdom,
and the one who gets understanding,
for the gain from her is better than gain from silver
and her profit better than gold."

Proverbs 3:13–14, Holy Bible (ESV)

Abstract

On Implicit Program Constructs

Session types are a well-established approach to ensuring protocol conformance and the absence of communication errors such as deadlocks in message passing systems.

Implicit parameters, introduced by Haskell and popularised in Scala, are a mechanism to improve program readability and conciseness by allowing the programmer to omit function call arguments, and have the compiler insert them in a principled manner at compile-time. Scala recently gave implicit types first-class status (implicit functions), yielding an expressive tool for handling context dependency in a type-safe manner.

DOT (Dependent Object Types) is an object calculus with path-dependent types and abstract type members, developed to serve as a theoretical foundation for the Scala programming language. As yet, DOT does not model all of Scala's features, but a small subset. Among those features of Scala not yet modelled by DOT are implicit functions.

We ask: can type-safe implicit functions be generalised from Scala's sequential setting to message passing computation, to improve readability and conciseness of message passing programs? We answer this question in the affirmative by generalising the concept of an implicit function to an *implicit message*, its concurrent analogue, a programming language construct for session-typed concurrent computation.

We explore new applications for implicit program constructs by integrating them into four novel calculi, each demonstrating a new use case or theoretical result for implicits.

Firstly, we integrate implicit functions and messages into the concurrent functional language LAST, Gay and Vasconcelos's calculus of linear types for asynchronous sessions. We demonstrate their utility by example, and explore use cases for both implicit functions and implicit messages.

We integrate implicit messages into two pi calculi, further demonstrating the robustness of our approach to extending calculi with implicits. We show that implicit messages are possible in the absence of lambda calculus, in languages with concurrency primitives only, and that they are sound not only in binary session-typed computation, but also in multi-party context.

Finally we extend DOT to include implicit functions. We show type safety of the resulting calculus by translation to DOT, lending a higher degree of confidence to the correctness of implicit functions in Scala. We demonstrate that typical use cases for implicit functions in Scala are typably expressible in DOT when extended with implicit functions.

Acknowledgements

I would like to thank my supervisor, Dr Martin Berger, for his patience, support and expertise throughout my research. I am confident that less patient and understanding supervisors would not have tolerated me. Thanks also to my thesis committee, Dr Ian Mackie and Dr Bernhard Reus for their helpful comments and advice.

I would like to thank my family for their encouragement and support throughout my PhD – my parents Paul and Amy Jeffery, my wife Sheila Jeffery, and my Grandmother Susan Haydon-Knowell, who have been there to help and listen whether things were going well or badly.

I would like to thank my colleagues in the PhD office for their friendship, and for making the office a fun and interesting place to come and work.

I would like to thank D. Castro, S. Gay, A. Scalas, V. Vasconcelos, M. Odersky, N. Amin, P. Giarrusso, J. Vitek, F. Křikava, M. Madsen, M. Rapoport, P. Schrammel, D. Orchard, and the anonymous reviewers of [Jeffery and Berger, 2018; Jeffery and Berger, 2019; Jeffery, 2019], who read my work and made it better by their comments.

Contents

List of Figures

List of Abbreviations

STLC	**S**imply **T**yped **L**ambda **C**alculus
LLC	**L**inear **L**ambda **C**alculus
HDMP	Lambda Calculus with **H**indley-**D**amas-**M**ilner **P**olymorphism
CCS	**C**alculus of **C**ommunicating **S**ystems
LAST	**L**ambda Calculus with **A**synchronous **S**ession **T**ypes
IM	LAST with **I**mplicit **M**essages
PLST	**P**i calculus with **L**inear channel types and **S**ession **T**ypes
PIIM	**P**LST w**i**th **I**mplicit **M**essages
MPST	**M**ulti**P**arty **S**ession **T**ypes
MPIM	**MP**ST with **I**mplicit **M**essages
DOT	**D**ependent **O**bject **T**ypes
DIF	**D**OT with **I**mplicit **F**unctions

To Dad

Chapter 1

Introduction

Computers process information in the form of *bits*, or binary digits – ones and zeroes – and therefore must be given instructions in a language of bits, usually called *binary*. Binary is extremely hard for human beings to read, and it is very hard for humans to write or modify computer programs written in binary. Programming languages were invented to mitigate this difficulty. Programming languages allow humans to express instructions to a computer in a human-readable form, and then translate their human-readable program code into binary so that a computer can execute it. The translation from human-readable code (*source code*) into binary is called *compilation*, and is performed by a program called a *compiler*. The first compiler had to be written in binary by an intrepid human programmer!

Programming languages have now existed for around 75 years [Knuth and Pardo, 1980]. Over that time, they have become increasingly sophisticated, allowing computer programmers to express more complex and detailed instructions to a computer with less source code. This greatly increases the speed at which complex programs with many features can be developed. Due to the complexity and difficulty of computer programming, even when using sophisticated programming languages, compilers have been augmented with error detection mechanisms that can detect some kinds of error in the programmer's source code before the code is translated into binary or executed by the computer.

A well-known category of errors in source code programs are *type errors*, where an operation intended to manipulate data is applied to data of the wrong type, for example, arithmetic operations on strings of text. To mitigate this kind of error, modern programming languages assign *types* to programs or program parts, that describe the behaviour of those programs or program parts. Compilers then use sets of rules, often called *type systems*, that describe the ways in which programs of certain types can be combined, on the (sometimes justified) assumption that if these rules are not violated, then type errors cannot occur. These type systems are often proved mathematically to be correct. These proofs are known as *type soundness proofs*, and usually relate to idealised versions of real programming languages known as programming language *calculi*, because real

programming languages are generally too complex, with too many features, which make type soundness proofs extremely difficult or impossible.

This thesis concerns a class of programming language feature known as *implicits*, which are designed to simplify the task of computer programming by leveraging compilers to analyse source code, using types to 'fill in' missing parts of the source program, reducing the workload of the programmer and improving the readability of source code. The process of the compiler filling in implicit parts of the program (making them explicit) is often called *implicit resolution*.

1.1 Thesis outline

The remainder of this chapter outlines background work relating to the research content of this thesis (section 1.2), the main topics covered are the aforementioned programming language constructs called implicits; *session types*, a type system that helps prevent errors in programming languages that allow concurrent execution of programs that communicate with each other; and the popular programming language *Scala* and its theoretical foundations (the calculus *DOT*). We then introduce our work in brief, summarising our contribution to the research field of programming languages (section 1.3).

Chapter 2 introduces in greater detail the context and background to the research content in this thesis. Chapters 3, 4 and 5 introduce theoretical foundations for programming languages with a novel implicit program construct called *implicit messages*, a feature for concurrent programming languages with session types. Chapter 6 lays down more solid theoretical foundations for the programming language Scala by integrating implicit functions, an implicit program construct popularised by Scala, into the calculus DOT, which is a programming language calculus designed to serve as a theoretical foundation for Scala, but that previously did not provide a foundation for Scala's implicit functions. The calculi introduced in chapters 3, 4, 5 and 6 are all accompanied by type soundness proofs. Chapter 7 concludes with some critical analysis of the research, discussing possible future work and key related work.

1.2 Background

1.2.1 Implicits

Implicits are a class of semi-related programming language constructs, in which the programmer can omit parts of a program that are otherwise essential, and the compiler infers those parts of the program automatically. There are three major types of implicits: *implicit*

parameters and *implicit conversions* which have found success in the programming language Scala; and the novel *implicit messages* [Jeffery and Berger, 2018; Jeffery and Berger, 2019], which are the main topic of this thesis.

Implicit parameters

Implicit parameters [Lewis et al., 2000; Oliveira et al., 2012; Odersky et al., 2018] are the most widespread and well known kind of implicit program construct. They are a direct generalisation of *default* parameters, in which function definitions can provide *default values* for their arguments. If the programmer then decides to omit those arguments at call-sites, the compiler will fill in the missing parameters with the defaults provided at the definition. Implicit parameters are similar – instead of providing a default value at the definition site, the programmer simply marks some arguments as implicit at the definition site, which communicates to the compiler that it may need to fill in that parameter at call sites, without saying what it should fill it in with. The language also provides a way to mark *variables* as implicit. When the compiler finds a call-site with a missing implicit parameter, it looks for a nearby variable of the appropriate type marked as implicit, and writes that variable in as the missing parameter.

Implicit function types

Implicit parameters as thus far introduced have not affected the types of functions that have implicit parameters. If we take a function and make one of its explicit parameters implicit, its type does not change[1]. Implicit function types [Odersky et al., 2018] are a generalisation of implicit parameters that lift implicits to the level of the function's type, breaking the property that the implicitness or explicitness of parameters do not affect types. With implicit function types, the type of a function communicates which of its parameters are implicit. This generalisation allows for greater flexibility, giving more of the programmer's work to the compiler. Implicit parameter names can be omitted entirely, as the function's type already communicates that the compiler should find an implicit parameter. A new operation known as *implicit query* (or simply *query*) is added to the language. The compiler replaces the query operations with the passed implicit parameters. Scala's query operation is written `implicitly[T]` where the type `T` is the type of the implicit parameter to be inserted in place of the query. The example Scala code below illustrates the difference between implicit parameters and implicit function types.

[1] Note that this holds for Scala, but not Haskell. This introduction focuses on implicits à la Scala.

```
// Standard implicit parameter
def f(x: A, y: B, implicit z: C): D = f(x, g(y, z))

// Implicit function type and query
def f(x: A, y: B): implicit C => D = f(x, g(y, implicitly[C]))
```

Implicit parameters and implicit function types can form a foundation for generic programming. Functions that take a generic parameter can implicitly take an extra parameter that describes how the generic parameter can be used.

Implicit conversions

Implicit conversions have appeared in several well-known programming languages such as C++, C#, Javascript and Scala under different names. They are a type-level feature in which unary functions are marked as implicit. If a type mismatch occurs in the program, the compiler tries to fix the type mismatch by finding an implicit conversion from the type of the mismatched value to a type that would match, and wrapping the mismatching value in a call to the implicit conversion. For example, if a function expects an integer as an argument, but receives a boolean, it will look for an implicit conversion (a unary function marked as implicit) from booleans to integers, and insert a call to that conversion between the function and the argument. The described example is shown below as Scala code.

```
// The implicit conversion
implicit def boolToInt(b: Boolean): Int = if (b) 1 else 0

// A function expecting an Int
def addTen(x: Int): Int = x + 10

// An erroneous call
addTen(true)

// The erroneous call after the compiler carries out
// implicit conversion
addTen(boolToInt(true))
```

Implicit conversions are perhaps responsible for the reputation that implicit program constructs have of making programs hard to understand. Implicit conversions in C++ and Javascript can easily obscure the meaning of programs due to the fact that they are global in scope, and enabled by default. In contrast, Scala's implicits are scoped and disabled by default.

Implicit conversions are not closely related to other kinds of implicit program construct and are not a primary topic of this thesis.

1.2.2 Concurrency

Pi calculus

Pi calculus [Milner, Robin and Parrow, Joachim and Walker, David, 1992] is a theoretical model of programming languages with first class message-passing concurrency, i.e. programming languages with built-in syntax for exchanging messages between concurrently executing threads or programs. The pi calculus reduces computation to three main behaviours, which can be executed in parallel and repeated ad infinitum: sending of messages; receiving of messages; and generation of communication channels. Pi calculus programs may experience faults such as deadlock, protocol mismatch, or errors due to a sender sending a message of a type that the receiver does not expect.

Session types

Session types [Honda, Vasconcelos, and Kubo, 1998; Honda, Yoshida, and Carbone, 2008] are a type system for message-passing concurrent programming languages like pi calculus, or concurrent lambda calculus. Session types ensure that every send operation is matched with a corresponding receive operation, where both sides expect the same type of message to be exchanged. Session types guarantee freedom from the classes of error mentioned in the previous paragraph. Session types were first formulated in the context of binary communication, typing protocols with two participants, but can be extended: multi-party session types can type protocols involving multiple parties. The safety guarantees provided by binary session types to two-party protocols hold for many participants with multi-party session types.

Implicit messages

Implicit messages [Jeffery and Berger, 2018; Jeffery and Berger, 2019], a contribution of this thesis, are an implicit program construct for message-passing concurrent programming languages with session types. Implicit messages allow the receiver of a message in a session-typed protocol to specify that the message is received implicitly, meaning that the sender of the message may omit the sending of the message, and the compiler deduces from the sender's session type that it should insert the corresponding send operation at the correct point in the protocol. The chosen message payload is a value marked implicit in the scope of the sender.

Implicit messages follow a similar logic to implicit function types – with implicit function types, two things are marked as implicit: the function definition and the implicit variable; and one thing is omitted (and later inserted): the implicit variable at the call site. With implicit messages, two things are marked implicit: a receive operation and the

implicit variable; and one thing is omitted (and later inserted): the corresponding send operation (sending the implicit variable).

The two-party protocol below shows a simple example of an implicit message, in a Scala-like syntax. The compiler can either use a session type given by the programmer to insert the missing message, or in a type-inference style situation, deduce the missing message by comparison to the receiver.

```
// Party 1
val x = 10; implicit receive y; send x + y

// Party 2
implicit val y = 10; receive z; print z

// Party 1, post-resolution
val x = 10; receive y; send x + y

// Party 2, post-resolution
val y = 10; send y; receive z; print z
```

In this thesis, we show that implicit messages can support generic programming and dependency injection in a concurrent context, just as implicit parameters and implicit function types support generic programming and dependency injection in a sequential context. Where information on how a generic parameter can be used, or contextual information, can be implicitly passed in a function call, such information can be passed with an implicit message in a concurrent setting. Implicit messages can be formulated in binary or multi-party session typed setting.

1.2.3 Scala

Scala is a general-purpose multi-paradigm programming language that facilitates functional and object-oriented styles of programming. It is perhaps best known for popularising implicit parameters/implicit function types and implicit conversions. While there is some well-founded criticism that Scala's implicits make programs hard to understand or modify, the frequency with which they appear in Scala programs suggest that programmers find these features useful nonetheless [Křikava, Vitek, and Miller, 2019].

Dependent Object Types (DOT)

DOT is a programming language calculus intended to be a theoretical model of Scala. DOT models a small subset of Scala's features, proving correctness properties for those features. The main features of Scala that DOT has covered with correctness guarantees are *fully path-dependent types* [Rapoport and Lhoták, 2019] and *abstract type members* [Amin, Moors, and Odersky, 2012].

DOT with Implicit Functions (DIF)

A key feature of Scala that DOT did not model until recently is implicit functions. DIF (**D**OT with **I**mplicit **F**unctions) [Jeffery, 2019] is a calculus that extends DOT with implicit functions, with correctness guarantees.

1.2.4 Linearity

Linearity is a property of some type systems that enforces a rule that variables must be used exactly once [Walker, 2005]. The enforcement of this property allows the guarantee of certain useful properties, such as the inability to use a file that has been closed, or to free unneeded memory more than once (both of these properties prevent common programming errors).

A drawback to linear languages is that any variable that is needed multiple times must be *rebound* after every use if it is needed again. This means that all functions that use the variable must return a reference to it so that it can be reassigned to a new variable after the call returns. This adds syntactic noise to programs and inhibits their readability. There are several solutions to this *rebinding problem* that programming languages can employ. One such solution, and a contribution of this thesis, is to use implicit parameters to hide rebinding.

1.3 Contributions

This thesis makes the following research contributions:

- We introduce the concept of implicit messages, and develop the first three programming language calculi that support them, namely:
  - IM (chapter 3, [Jeffery and Berger, 2018]), a concurrent, session-typed lambda calculus that also includes implicit functions, based on a concurrent lambda calculus known as LAST.
  - PIIM (chapter 4), a binary session-typed pi calculus.
  - MPIM (chapter 5, [Jeffery and Berger, 2019]), a multi-party session-typed pi calculus.
- We introduce the *session type class pattern*, a form of generic programming, and the *concurrent dependency injection pattern*, a form of dependency injection. Both patterns are for session-typed concurrency and enabled by implicit messages (see chapter 3).
- We add implicit functions to two calculi: LAST, resulting in the calculus IM; and DOT, resulting in the calculus DIF (chapter 6, [Jeffery, 2019]), which:

- In IM allows us to demonstrate a novel solution to the well-known rebinding problem of linear languages (chapter 3).
- In DIF, provides a stronger theoretical foundation for Scala, by extending the feature set of Scala that DOT models. It is shown that common Scala patterns involving implicits can be modelled by DOT when it is extended with implicit function types.

Chapter 2

Literature Review[1]

2.1 Introduction

In this chapter we review in detail existing literature on topics related to this thesis. We begin with an overview of type systems, and proceed to look at specific programming languages and calculi that lay foundations for the novel calculi presented in this thesis. Calculi studied in this chapter include lambda calculi (section 2.3) and calculi for concurrency (section 2.4), especially pi calculi, which together lay foundations for the concurrent lambda calculus LAST that serves as a basis for our novel language IM (introduced in chapter 3), pi calculi also serving as a basis for the variants of IM, namely PIIM (introduced in chapter 4) and MPIM (introduced in chapter 5). Studied calculi also include the object calculus DOT (relevant for chapter 6), the foundation for the popular programming language Scala. We look at various features and type systems that can extend these calculi, especially type classes and implicits. Where appropriate we discuss real systems that implement those type systems and features.

2.2 Type Systems

Types are mathematical descriptions of computer programs or computer program fragments. They can express a plethora of properties that programs can exhibit, which may be useful to the program's developer or user, such as the nature of values dealt with by the program, e.g. dates, numbers and text, whether the program makes use of a network connection, and if so, what kind of information is exchanged with the network, or whether the program terminates or loops indefinitely. A *type system* is any formalism, theoretical or actual, that assigns types to programs. Type systems for classification of sequential programs are an integral part of modern programming languages used in industry, and have been a topic of academic research for, arguably, at least a century

[1]Small portions of this chapter are adapted from [Jeffery and Berger, 2018] and [Jeffery and Berger, 2019], published works co-authored with my supervisor, Dr. Martin BERGER, and [Jeffery, 2019], which is published original work. I estimate that 95% of this chapter is completely my own work, with the last 5% being co-written.

[Russell and Whitehead, 1910–13]. Type systems for concurrent, parallel and distributed computing, often referred to generally as *behavioural* type systems, are a more recent and contemporary research area.

2.2.1 Motivation

Type systems are such a critical aspect of the modern software industry, and a huge research area, because they can be used to identify programs that exhibit properties of interest. They can be a useful tool in determining if a program behaves as its writer intended it to, or if it behaves in a manner that might make it liable to produce an erroneous result. Given the ubiquity of software in modern society, the investigation of type systems should then be of great value. Programs that perform ill-defined operations, fail to terminate, function insecurely[2], can more easily be identified with the aid of type systems. Any given type system must be associated with a specific programming language, and the nature of the type system is fundamentally linked to its language. The kinds of errors a program can exhibit are determined by the nature of the language the program is written in and therefore influenced by the programming language's type system. A problem with a program may be addressed by thinking about a programming language and type system that identify programs that exhibit the problem.

While there is debate as to when best to use type systems to analyse programs (primarily between proponents of analysis during the development phase (*static* typing) versus those of analysis during the testing phase (*dynamic* typing) of the software development process), and indeed there are large shortcomings in the capabilities of type analysis - for example, it is impossible to tell, in the most general case, if a program will terminate or loop indefinitely - few or none argue against the use of type systems in the software development process.

2.2.2 Types of Type System

Type systems are often named in relation to the program properties they guarantee, or, conversely, the class of errors that they prevent. We will now look at several kinds of type system, and will see that they relate to classes of errors that might occur in programs.

A type system generally makes two additions to the specification of a programming language: (1) a grammar of types, which describe the types of values that functions accept as arguments and return; and (2) a formal system based on *logical judgements*, usually written $\Gamma \vdash M : T$ or similar, that express the type T of a program M under assumptions Γ regarding the types of the free variables in M. They are true or false statements about the validity of assigning the type T to the program M, and are judged true of false by

[2] [Church, 1940], [Martin-Löf, 1975], and [Schneider, Morrisett, and Harper, 2001] discuss type systems that address these properties respectively.

an accompanying set of inference rules that allow judgements about complex programs to be built from simpler ones. We now look briefly at some well known types of type system.

Polymorphic

Polymorphic type systems are those that allow programs or program parts to have multiple types, or facilitate reuse of code by allowing code parameterisation by values of multiple types. *Parametric* polymorphism achieves this by allowing a single piece of code to truly be executed over parameters of different types. In contrast, *ad-hoc* polymorphism, often called *overloading*, allows the programmer to specify different but synonymous behaviours for values of different types. These kinds of polymorphism were first identified in 1967 in [Strachey, 2000]. Another well-known type of polymorphism is *Subtype* polymorphism, which allows the programmer to construct a hierarchy of related types. Subtype polymorphism first appeared in the programming language Simula.

Parametric. Parametrically polymorphic code truly accepts arguments of different types. The simplest parametrically polymorphic program is the identity function, shown below in Haskell (left) and Java (right).

```
id :: a -> a
id x = x
```

```
<A> A id(A x) { return x; }
```

Parametric polymorphism is ideally suited to developing generic collection libraries. Functions can be written that work, for example, for all lists, regardless of what types are contained in the list. Prototypical examples of such functions are: `head`, which returns the first element of a list; `tail`, which returns all elements in a given list except the first; `append`, which, given an element and a list, returns a new list containing the elements from the given list followed by the element; and `concat`, which, given two lists, returns a new list containing all elements from the first list followed by all elements from the second list. Shown below are example implementations of these functions in Haskell.

```
head :: [a] -> a
head l = case l of { [] -> undefined ; (h:t) -> h }

tail :: [a] -> [a]
tail l = case l of { [] -> undefined ; (h:t) -> t }

append :: a -> [a] -> [a]
append e l = case l of { [] -> [e] ; (h:t) -> h:(append e t) }

concat :: [a] -> [a] -> [a]
concat l1 l2 = case l2 of { [] -> l1 ; (h:t) -> concat (append h l1) t }
```

A consequence of parametric polymorphism is that since code must work for all types, it necessarily cannot interact with its polymorphic parameters in non-trivial ways. For example, a parametrically polymorphic function may not add two parameters together, since, while the function may work for integers and floating point numbers, it would fail when given arguments of types for which there is not a notion of addition.

Ad-hoc. Languages with Ad-hoc polymorphism allow the user to define multiple functions of the same name, which vary in the type (and sometimes the number) of arguments (sometimes called *overloading*). Calls to such functions are disambiguated either at compile-time, in the case of statically typed languages, or at runtime with dynamic languages. In Haskell, ad-hoc polymorphism is mediated by *type classes*, which constrain the types of overloaded functions using a *class definition*. These allow the types of overloaded functions to vary in a single type parameter. An example class definition is show below:

```
class Add n where
    add :: n -> n -> n
```

This definition states that the function `add` can be overloaded by providing an instantiation of the type variable `n` using an *instance definition*. Two examples are shown below, one for Peano-style natural numbers and one for integers:

```
instance Add Nat where
    add Zero x = x
    add (Succ x) y = add x (Succ y)

instance Add Int where
    add x y = x + y
```

A function application `add` e_1 e_2 is *resolved* at compile time with either `Nat` or `Int`'s version of `add`, depending on the types of e_1 and e_2.

Other languages allow the types of overloaded functions to vary more freely. For example in Java, we may use arbitrary types in functions of the same name[3]:

```
int convert(String s) { return Integer.parseInt(s); }
String convert(int x) { return "" + x ; }
```

Subtype. Subtype polymorphism concerns hierarchical relationships between types defined by the programmer. For example, they might define a type `Vehicle`, and types `Car` and `Aeroplane`, such that `Car` and `Aeroplane` are subtypes of `Vehicle`, allowing `Car`s and `Aeroplane`s to support all operations that `Vehicle`s support, with the reverse not being the case - `Car`s would have their own unique operations, as would `Aeroplane`s, but they would have all `Vehicle` operations in common. The example

[3]This example can also be expressed in Haskell using *type families*.

below illustrates subtype polymorphism, with allowable and erroneous operations indicated by `// comments`.

```
class Vehicle { abstract def speed(): Int }
class Car extends Vehicle { def speed = 30; def drive(): Unit = ...  }
class Aeroplane extends Vehicle { def speed = 250; def fly(): Unit = ... }
print(Car.speed() + Aeroplane.speed()) // ok, prints 280
Aeroplane.fly() // ok
Car.fly() // error
Car.drive() // ok
```

Dependent

Dependent type systems are those that allow for the presence of values in types. Thus far, there has been a clear separation between type and value, but with dependent types, this line blurs. For example, a type for lists may be augmented to also contain the length of the list as part of the type. An example is shown below in the dependently typed language Agda [Norell, 2008]:

```
data List(A: Set): Nat → Set where
    []   : List A 0
    _::_ : ∀{n} → A → List A n → List A (1 + n)

concat : ∀{A m n} → List A m → List A n → List A (m + n)
concat [] l = l
concat (e :: l1) l2 = e :: concat l1 l2
```

The `data` definition defines a new data type called `List`, which takes a type parameter `A` and a value parameter of type `Nat`. The constructor `[]` builds an empty list, with type `List A 0` - note the value `0` in the type expression. The constructor `_::_` builds a list from an element and a list, the element having type `A`, and the argument list type `List A n`. The resulting list differs in type to the input list, in that the value `n` is one greater, capturing the increase in length caused by the addition of a new element. The function `concat` shows similar behaviour for combining two lists - the length value in the type of the output list is the sum of those for the input lists.

Dependent type systems allow the compiler to determine at compile-time that no out-of-bounds list accesses occur, without the need for runtime bounds checks. They also allow for the expression of more complex properties in types, such as types for sorted lists. These type systems typically force termination and are therefore not *Turing-complete*, meaning that they cannot express all computations expressible by a Turing machine, in this case specifically with regards to termination. The use cases of dependently typed programming languages are generally in verified programming and theorem proving, and in such cases guaranteed termination is desirable.

Behavioural

Behavioral types are those that constrain not only the values that are passed around a program, but also the control-flow of the program itself. The best-known variant of behavioural type systems are session types [Honda, Vasconcelos, and Kubo, 1998]. Session types are types for message-passing concurrency, which constrain the sequence of messages that can be exchanged between communication partners in order to prevent communication errors. For example, a client is prevented from sending a string to a server at such a time as the server expects to receive an integer from the client. We will look at session types in more detail in section 2.4.

2.3 Lambda Calculus

The lambda calculus was conceived in the 1930's as a foundational calculus for logic, by Alonzo Church [Church, 1932]. It proved unsuccessful in this regard, but was nonetheless adopted as a foundational calculus for functional programming languages, and due to its Turing-completeness, is suitable as such. Widely used, modern programming languages such as Haskell, ML, Racket and Clojure are based on the lambda calculus. It serves as a base for theoretical models of programming languages, allowing for proofs of programming language correctness. It can be extended with type systems, allowing for the identification of programs with particular properties of interest. We now look at the lambda calculus in detail, and study some of its applications relevant to the topic of this thesis.

2.3.1 Untyped Lambda Calculus

The untyped lambda calculus is extremely simple, with only three kinds of terms. Abstractions $\lambda x.M$ represent a function whose argument name is x and body is M. Applications $M\ N$ represent the application of the function M to the argument N. Variables x refer to the argument applied to the abstraction that binds them. We specify formally the *context-free grammar* (sometimes referred to simply as the *grammar* or the *syntax*) of lambda calculus, which specifies recursively the structure of lambda calculus terms (programs) in figure 2.1.

$$
\begin{array}{lcll}
M, N & ::= & \lambda x.M & \text{(ABSTRACTION)} \\
 & | & M\ N & \text{(APPLICATION)} \\
 & | & x & \text{(VARIABLE)} \\
x & ::= & \textit{(variable names)} &
\end{array}
$$

FIGURE 2.1: Syntax of Lambda Calculus

Figure 2.2 gives the call-by-value *semantics* of the lambda calculus. Semantics specify how to *evaluate* terms. Evaluating a term corresponds to executing a program. We consider the term left over after evaluating a term to be the result of evaluating it (the output of the program), in the same way that the result of evaluating the term $1 + 2$ is the term 3. With call-by-value semantics, we may only substitute an argument into a function when it has been reduced to a value (sometimes called *normal form*). The rule [β-RED] captures this, and performs *capture avoiding substitution*, replacing occurrences of the bound variable x in the function body M with the function argument $\lambda y.N$. This is called β-reduction. β-reduction is permitted only when the argument is reduced to an abstraction, which is the essence of call-by-value. The rules [APPL] and [APPR] allow reduction of function and argument expressions respectively in application.

Substitution notation $[^M/_x]$ denotes a function that replaces all free occurrences of the variable x in its argument by the term M in a capture avoiding manner. Capture occurs when we substitute a term M with free variables into another term N, such that a lambda expression within N binds a name matching one of the free variables in M, such that the lambda in N will then appear to bind one of the previously free variables in M. Capture avoiding substitution renames bound variables and their binders in the argument term N such that capture does not occur, and the meanings of terms are preserved – that is, the typing and semantics of the terms are not changed. All references to the operation of substitution in this thesis can be assumed to refer to capture avoiding substitution, unless otherwise stated.

$$\frac{M \to M'}{M\ N \to M'\ N}\,[\text{APPL}] \qquad \frac{N \to N'}{M\ N \to M\ N'}\,[\text{APPR}]$$

$$(\lambda x.M)N \to M[^N/_x] \text{ if } N = \lambda y.N' \text{ or } y \quad [\beta\text{-RED}]$$

FIGURE 2.2: Call-by-Value Semantics of Lambda Calculus

Call-by-name semantics are similar to call-by-value semantics, but we disallow reducing arguments in application expressions prior to substitution, but β-reduction for terms whose arguments are not in normal form is permitted. The call-by-name semantics of lambda calculus are given in figure 2.3.

$$\frac{M \to M'}{M\ N \to M'\ N}\,[\text{APPL}] \qquad (\lambda x.M)N \to M[^N/_x] \quad [\beta\text{-RED}]$$

FIGURE 2.3: Call-by-Name Semantics of Lambda Calculus

Most programming languages' semantics are closer to call-by-value than to call-by-name, although some are closer to call-by-name, most notably Haskell. Call-by-value is thought to be more efficient for typical styles of programming, but fails to terminate for a larger set

of programs than call-by-name. Languages with call-by-name therefore more naturally express infinite data structures such as streams.

Below is an example of a lambda calculus term that fails to terminate with both call-by-name and call-by-value semantics. We see that after a single β-reduction, we obtain a term identical to the original term, and the reduction is therefore nonterminating.

$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow (\lambda x.x\ x)[^{(\lambda x.x\ x)}/_{x}] \equiv (\lambda x.x\ x)(\lambda x.x\ x)$$

A problem with the untyped lambda calculus is that it can express programs that do not terminate, or reach a state in which they are not in normal form, yet no reduction rule applies. We will see in the next subsection how this problem can be mitigated.

2.3.2 Simply Typed Lambda Calculus

The simply typed lambda calculus (STLC) [Church, 1940] imposes a simple type system on lambda calculus, preventing the aforementioned problems, yet it is arguably a less practical programming language due to the property of strong normalisation and therefore lost Turing universality. Strong normalisation is the property that all terms reduce to a single normal form. A strongly normalising language cannot express arbitrary computations.

STLC's additions to the grammar of lambda calculus are shown in figure 2.4. These additions are types T, which are either functions $T \rightarrow T$, or base types $\mathbf{B}$, e.g. unit, booleans, integers. Depending on the base types, we also extend the grammar of terms with constants $\mathbf{C}$ to inhabit base types. In the simplest case, we have $\mathbf{B} ::= \mathbf{Unit}$ and $\mathbf{C} ::= \mathbf{unit}$.

$$\begin{array}{lcll} M, N & ::= & \ldots \mid \mathbf{C} & (\text{CONSTANTS}) \\ T & ::= & T \rightarrow T & (\text{FUNCTIONS}) \\ & \mid & \mathbf{B} & (\text{BASE TYPES}) \end{array}$$

FIGURE 2.4: Grammar of types for Simply Typed Lambda Calculus

In untyped lambda calculus, there are terms that are not in normal form but cannot be evaluated, such as $x\ y$. However in the correct context, these terms can be evaluated - consider the term $(\lambda x.x\ y)(\lambda x.x)$ containing the previous example as a subterm. The operational semantics of STLC are identical to that of untyped lambda calculus (when we consider constants $\mathbf{C}$ to be terms in normal form). There are terms expressible in the grammar of STLC that are not in normal form and cannot be evaluated in any context (because no reduction rule applies to them), for example **unit unit**. In the untyped lambda calculus, we were not concerned with such errors, but in STLC we seek to prevent them. Such terms are said to be *stuck*. It is desirable to identify the set of STLC terms that are

not stuck and do not get stuck after some evaluation. To this end STLC introduces *typing rules*. These are logical rules that determine whether or not a term gets stuck. If, by using these rules, we can derive the statement $\Gamma \vdash M : T$, then the term M does not get stuck in a context where its free variables are bound to values whose types are given by Γ, but is guaranteed to evaluate to a value of type T. We give typing rules for STLC in figure 2.5. Figure 2.6 shows how these rules are applied in an example proof for the typing of a simple term.

$$\frac{\Gamma \vdash M : T \to U \quad \Gamma \vdash N : T}{\Gamma \vdash M\ N : U}\ [\textsc{App}] \qquad \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \to U}\ [\textsc{Abs}]$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ [\textsc{Var}] \qquad \frac{-}{\Gamma \vdash \textbf{unit} : \textbf{Unit}}\ [\textsc{Const}]$$

FIGURE 2.5: Typing rules for Simply Typed Lambda Calculus

$$\dfrac{\dfrac{\dfrac{y : T \in \{x : T, y : T\}}{x : T, y : T \vdash y : T}\ [\textsc{Var}]}{x : T \vdash \lambda y.y : T \to T}\ [\textsc{Abs}] \qquad \dfrac{x : T \in \{x : T\}}{x : T \vdash x : T}\ [\textsc{Var}]}{x : T \vdash (\lambda y.y)\ x : T}\ [\textsc{App}]$$

FIGURE 2.6: Example typing proof using STLC typing rules

In order to have certainty that typable terms do not get stuck, we need a *type soundness proof*. This is a formal proof of the theorem $\Gamma \vdash M : T \Rightarrow M \to^* v$, where v is a term in normal form. It tells us that the typing rules function as intended. Such proofs are generally achieved by induction on typing rules, and are required for any programming language and type system pair in order to have guarantees about the correctness of the type system, and therefore that typable programs exhibit the desired properties. The exact nature of the theorem depends on what properties the type system is intended to guarantee. In the case of languages that can perform I/O, we may want to guarantee that all I/O interactions are *safe*, in the sense that any sent data is in the format that the receiver expects, or other properties. These can be reflected in the type soundness theorem and justified in the type soundness proof. We will see examples of these in later chapters of this thesis.

A shortcoming of typing systems such as that of STLC is that while they disallow all terms that get stuck, they do not permit all terms that do not get stuck. For example the term $(\lambda x.y)(\textbf{unit}\ \textbf{unit})$, is not typable, but evaluates safely to y. Such a limitation is a consequence of Rice's Theorem [Rice, 1953] – if we had a type system that allowed exactly those terms that do not get stuck for a Turing-complete language, we could decide the halting problem, and such a type system is therefore impossible.

2.3.3 Hindley-Damas-Milner Polymorphism

A shortcoming of STLC is the lack of polymorphism which leads to duplication of code. In order to work with typed lists, it is necessary to add types `List Bool`, `List Int`, `List(Int` $\rightarrow$ `Int)`, etc. The problem is that it then becomes necessary to implement similar (or even identical) list functions once for each type that is used inside a list. For example, to implement an indexing function that retrieves the element at a given index from a given list, we would need to implement this function *once for every type* that we want to use within lists, which is extremely repetitive and laborious, and makes code difficult to modify. This is a consequence of STLC's typing system which enforces that if we use a function on a value of a given type, the function can only be reused on values of that type.

Hindley-Damas-Milner Polymorphism [Hindley, 1969; Damas and Milner, 1982] gives us a solution to the code duplication problem of STLC by adding *type variables*, and *type schemas*, which are essentially binders for type variables (in the same way that lambdas are binders for variables at the term level). Type variables and type schemas allow us to keep a type within some code abstract, and instantiate the abstracted type in a single piece of code with different types at different points in the program. This is a form of parametric polymorphism. As a result, lambda calculus with Hindley-Damas-Milner Polymorphism (HDMP) is much more usable than STLC for non-trivial programs. Indeed it forms the basis for many industrial strength programming languages such as ML.

We extend the grammar of STLC types to include type variables. We also add type schemas, which are binders for type variables, and are not mixed freely with types, but always at the 'outermost level' of a type, so, for example, $\forall x.(\texttt{Bool} \rightarrow x)$ is allowed, but $\texttt{Bool} \rightarrow \forall x.x$ is not. Although not strictly required for HDMP, we introduce a list type (and corresponding constants) to our presentation, to better illustrate the applications of HDMP. The syntax additions of HDMP over STLC are given in figure 2.7 and the type schemas for constants are given in figure 2.8.

The typing system for HDMP adds rules for the key new constructs. The rule [LET] types let-bindings by checking that the use of the bound expression M is used sensibly in the body expression N. The rule [GEN] allows typing an expression M as though it had type S when the name α bound in the scheme $\forall\alpha.S$ is irrelevant. The rule [INST] allows using a more general type in place of a more specific one, wherein the judgement $S > S'$ reflects that S can be obtained from S' by substituting concrete types for some of the type variables bound by S' and is therefore a more specific version of S. An example of this would be using `List Bool` where $\forall\alpha.$ `List` α is expected. New typing rules for HDMP are given in figure 2.9 – STLC rules in figure 2.5 also apply.

The for-all types of HDMP allow for highly reusable code. We can write generic functions over collections e.g. lists, that can be used (and reused) on lists containing

$$
\begin{array}{lcll}
M, N & ::= & \ldots \mid \mathbf{C} & \text{(CONSTANTS)} \\
 & \mid & \texttt{let}\ x = M\ \texttt{in}\ N & \text{(LET-BINDING)} \\
 & & & \\
T & ::= & \ldots \mid \alpha & \text{(TYPE VARIABLES)} \\
 & \mid & \texttt{List}\ T & \text{(LISTS)} \\
 & & & \\
S & ::= & \forall \alpha.S & \text{(TYPE QUANTIFIERS)} \\
 & \mid & T & \text{(UNQUANTIFIED TYPES)} \\
 & & & \\
\mathbf{C} & ::= & \texttt{nil} \mid \texttt{null} \mid \texttt{cons} \mid \texttt{hd} \mid \texttt{tl} & \text{(LIST CONSTANTS)} \\
 & \mid & \texttt{true} \mid \texttt{false} & \text{(BOOLEAN CONSTANTS)}
\end{array}
$$

FIGURE 2.7: Grammar of HDMP

$$
\begin{array}{rcl@{\qquad}rcl}
\texttt{nil} & : & \forall \alpha.\, \texttt{List}\ \alpha & \texttt{tl} & : & \forall \alpha.\, \texttt{List}\ \alpha \to \texttt{List}\ \alpha \\
\texttt{null} & : & \forall \alpha.\, \texttt{List}\ \alpha \to \texttt{Bool} & \texttt{true} & : & \texttt{Bool} \\
\texttt{cons} & : & \forall \alpha.\, \alpha \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha & \texttt{false} & : & \texttt{Bool} \\
\texttt{hd} & : & \forall \alpha.\, \texttt{List}\ \alpha \to \alpha & & &
\end{array}
$$

FIGURE 2.8: Type schemas for HDMP constants

$$
\frac{\Gamma \vdash M : S}{\Gamma \vdash M : \forall \alpha.S}(\alpha \notin fv(\Gamma)) \quad [\text{GEN}]
$$

$$
\frac{\Gamma \vdash M : S}{\Gamma \vdash M : S'}(S > S') \quad [\text{INST}] \qquad \frac{\Gamma, x : T' \vdash M : T' \quad \Gamma, x : T' \vdash N : T}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : T} \quad [\text{LET}]
$$

FIGURE 2.9: Typing rules for HDMP

elements of any one type. For example, if we assume standard integers and conditionals, we can use the list constants to build a function that will index into a list returning the n^{th} element:

```
let index = λl. λn. if n == 0 then hd l else index l (n - 1)
```

We can write `index (cons 2 (cons 3 (cons 5 nil))) 2` and `index (cons true (cons false (cons false nil))) 1` in the same program without the type checker rejecting the program. This is because the type of `index` is $\forall x.\ \texttt{List}\ x \to \texttt{Int} \to x$, which can be instantiated via the rule [INST] to both $\texttt{List Int} \to \texttt{Int} \to \texttt{Int}$ and $\texttt{List Bool} \to \texttt{Int} \to \texttt{Bool}$.

We can further make use of this polymorphism to create a generic sorting function. We achieve this by parameterising our function by a comparator function which captures the ordering of elements.

```
let isort = λl. λf.
    let insert = λx. λl. λf.
        if null l then cons x nil
        else if f x (hd l) then cons x l
        else cons (hd l) (insert x (tl l) f)
    in if null l then nil else insert (hd l) (isort (tl l) f) f
```

HDMP is, like STLC, strongly normalising, making it impractical for real-world programming. HDMP and STLC force termination (i.e. are strongly normalising) because they do not allow to type recursive functions. There are many ways to allow arbitrary computation in HDMP and STLC, and perhaps the simplest would be to add a fixed-point combinator to its syntax (not encoding it in other constructs, which is untypable).

HDMP restricts type schemas such that they may only appear at the 'outermost level' of a type, and cannot occur within a type expression, which, for example, forbids the type $\texttt{List}(\forall\alpha.T)$. If we do away with this restriction, we obtain a system called *fully polymorphic* lambda calculus [Hindley, 1969; Girard, 1972], sometimes called 'System F' or 'second-order lambda calculus'.

2.3.4 Type Classes

Type classes [Kaes, 1988; Wadler and Blott, 1989] are a mechanism to allow ad-hoc polymorphism in languages with parametric polymorphism. Parametrically polymorphic type variables may be *constrained*, such that they can only be instantiated by types over which a user-defined set of functions exist. Further functions can then be defined over all types for which the set of functions are defined. A typical example of a type class is Haskell's `Show`. In a program, we might want to use a function `print` to print values over many different types, for example we might want to write `print 1`, `print True` and `print [1, 2]` in the same program. To achieve this with type classes, we give `print` the following signature, shown in Haskell-style syntax:

```
print :: Show a => a -> IO ()
```

This signature tells us that where `a` is a type in the `Show` type class, `print` takes an `a` and returns `IO ()` (in Haskell, `()` is the unit type and types of the form `IO T` represent values of type `T` obtained as a result of performing I/O). The definition of the `Show` type class itself is given below:

```
class Show a where
    show :: a -> String
```

We call this a *class definition*. It tells us that a type `a` is in the `Show` type class when there exists a function `show` of type `a -> String`. We can then define this function for a given type, for example `Bool`, as below, with an *instance definition*:

```
instance Show Bool where
```

```
    show b = if b then "True" else "False"
```

And for `Int`, assuming a function `intToString :: Int -> String` that converts an `Int` to its `String` representation:

```
instance Show Int where
    show i = intToString i
```

We can also declare type constructors to be in a type class when their argument types are also in the type class. For example, we can declare that tuples of type `(a, b)` for all `a`, `b` are in `Show` when `a` and `b` are in `Show` thusly:

```
instance (Show a, Show b) => Show (a, b) where
    show (a, b) = "(" ++ show a ++ ", " ++
                                show b ++ ")"
```

With these instance definitions at hand, we can give a definition for `print`, which will satisfy the type checker for arguments of any type in the `Show` type class, as above. The definition is as follows:

```
print a = putStrLn (show a)
```

Type classes are usually implemented via a technique known as *dictionary passing* [Kiselyov, 2014]. Function definitions whose type contains a type class constrained type variable are adapted to take an additional parameter for each such type variable. That additional parameter is known as a *dictionary*, and is a tuple containing the type class function implementations for the type that instantiates the type class constrained type variable. When a type class declares just a single function, such as with `Show`, the dictionary is just a single function rather than a tuple of functions. Call sites for those function definitions are augmented to pass the dictionary of appropriate type - the compiler decides what the appropriate type is based on the parameter passed at the call site.

In the case of the `Show` type class above, we would expect the `Bool` instance definition to be translated into the following dictionary:

```
showBoolDict :: Bool -> String
showBoolDict = show where
    show b = if b then "True" else "False"
```

The `print` function is then translated to the following:

```
print :: (a -> String) -> a -> IO ()
print dict a = putStrLn (dict a)
```

Call sites `print b` where `b` is of type `Bool` are then rewritten as `print showBoolDict b`. This rewriting of terms and insertion of dictionary parameters means that type classes can be thought of as a kind of implicit program construct.

2.3.5 Implicit Parameters and Implicit Function Types

Modularity, a core concept in software engineering, is greatly aided by parameterisation of programs. Parameterisation has dual facets: supplying and consuming a parameter. A key tension in large-scale software engineering is between *explicit* (e.g. pure functional programming), and *implicit* parameterisation (e.g. global state). The former enables local reasoning but can lead to repetitive supply of parameters. Here is a simple example of the problem, in Scala-style syntax:

```
def compare(x: Int, y: Int)(comparator:
    Int => Int => Boolean): Boolean =
  comparator(x)(y)
...
compare(3, 4)(<=)
...
compare(17, 12)(<=)
...
```

Repeatedly passing functions like <= which are unlikely to change frequently, is tedious, and impedes readability of large code bases. Default parameters are an early proposal for mediating this tension in a type-safe way. The key idea is to annotate function arguments with their default value, to be used whenever an invocation does not supply an argument:

```
def compare(x: Int, y: Int)(comparator:
    Int => Int => Boolean = <=): Boolean =
  comparator(x)(y)
...
compare(3, 4)
...
compare(17, 12)
...
```

The compiler synthesises `compare(3, 4)(<=)` from `compare(3, 4)`, and `compare(17, 12)(<=)` from `compare(17, 12)`. The missing argument indicates to the compiler that the default <= should be used. Default parameters have a key disadvantage: the default value is hard-coded at the callee, and cannot be context dependent. Implicit arguments separate the *callee's declaration* that an argument can be elided, from the *caller's choice* of elided values, allowing the latter to be context dependent. The example below shows two calls to `compare` in different contexts, where the implicit comparison operator varies.

```
def compare(x: Int, y: Int)(implicit comparator:
    Int => Int => Boolean): Boolean =
  comparator(x)(y)
...
implicit val cmp = <=
compare(3, 4)
...
implicit val cmp = >
compare(17, 12)
...
```

In this example `compare(3, 4)` is rewritten as above, but `compare(17, 12)` becomes `compare(17, 12)(>)`, i.e. a different implicit argument is synthesised. The disambiguation between several providers of implicit arguments happens at compile-time using type and scope information. Programs where elided arguments cannot be disambiguated at compile-time are rejected as ill-formed. Hence type-safety is not compromised.

Implicit arguments are a strict generalisation of default parameters. They were pioneered in Haskell [Lewis et al., 2000], and refined in [Oliveira et al., 2012]. They were popularised as well as further refined in Scala [Odersky et al., 2018], being lifted to the type level. In Scala's new *Dotty* compiler [Odersky, 2017], in addition to standard function types, there is an *implicit function type*, a type for functions whose argument is provided implicitly. This lifting to the type level allows for improved abstraction and reuse over implicit functions (Note: we often refer to functions whose arguments are implicit as *implicit functions*).

Coherence

The concept of coherence for languages with implicit program constructs is introduced in [Schrijvers, Oliveira, and Wadler, 2017]. Such a language is said to be coherent when there exists exactly one possible choice of translation for *implicit resolution*, the process of 'translating out' the implicits from a program, as described above. In previous examples, there is always a single obvious choice of value to insert as an implicit parameter or type class dictionary, but this is not necessarily the case. In the simplest incoherent case, we might have two implicit values in scope, e.g.:

```
implicit val a = 1
implicit val b = 2
def printInt(implicit toPrint: Int): Unit = print(toPrint)
printInt
```

We could reasonably expect `printInt` to print either `1` or `2`. Scala avoids much incoherence with precedence rules for choice of implicit value. More deeply nested implicit definitions are preferred over less deeply nested ones, and more specific types (e.g. `Int`

`=> Int`) are preferred over more general ones (e.g. α `=>` α). Scala does exhibit some, more subtle cases of incoherence, however[4].

Haskell's type classes are coherent, in contrast to Scala's implicit functions. Haskell prevents the definition of *overlapping* instance definitions, e.g. one cannot write `instance A (a -> a)` and `instance A (Int -> Int)` in the same program as `Int -> Int` is an instance of the more generic type `a -> a` and the instances therefore overlap. This restriction can be inconvenient, and so there is an option to disable this check in the compiler, but this breaks coherence guarantees. Haskell's `ImplicitParams` extension is coherent – its implicit parameters are resolved purely based on names, rather than selecting from all implicit values of the appropriate type. Therefore there is only ever a single candidate for insertion.

2.3.6 Type Classes via Implicits

Languages with implicit functions can leverage their implicit functions to mimic the behaviour of type classes [Oliveira, Moors, and Odersky, 2010]. Type classes are implemented via the insertion by the compiler of a dictionary argument to functions that take parameters whose types are constrained by type classes. Values akin to these dictionaries can be passed via implicit functions in languages that have implicit functions but lack type classes. This is best understood by example.

Consider a Haskell-style type class for collections:

```
class Collection c where
    add :: a -> c a -> c a
    singleton :: a -> c a
    remove :: c a -> Maybe (c a, a)
```

We could instantiate such a type class for both lists and sets[5] thusly:

```
instance Collection List where
    add a l = a : l
    singleton a = [a]
    remove l = if l == [] then Nothing else Just (tail l, head l)

instance Collection Set where
    add a s = insert a s
    singleton a = insert a empty
    remove s = if s == empty then Nothing else Just (deleteFindMin s)
```

With these instances at hand, we can write code over all `Collection`s. And example function over `Collection`s might be:

[4] This is the case at least as recently as Scala version 2.11.

[5] Note that Haskell's `Data.Set` type introduces an `Ord` type class dependency, so, for example, `Set.add` has type `Ord a => a -> Set a -> Set a` - we assume a similar `Set` API without this dependency, so would have the simpler type `a -> Set a -> Set a` for `Set.add`.

```
getTwo :: Collection c => c a -> Maybe (c a, a, a)
getTwo c = do (c' , a ) <- remove c
              (c'', a') <- remove c'
              return (c'', a, a')
```

At compile-time, the `class` definition is translated into a dictionary type - essentially a tuple containing functions whose types match those declared above:

```
type CollectionDict c a =
    ( a -> c a -> c a
    , a -> c a
    , c a -> Maybe (c a, a)
    )
```

Instantiations of the type class are then converted into dictionaries, whose types match `collectionDict c a`, where the instance type replaces the parameter `c`.

```
collectionListDict :: CollectionDict List a
collectionListDict =
    ( \a l -> a : l
    , \a -> [a]
    , \l -> if l == [] then Nothing else Just (tail l, head l)
    )

collectionSetDict :: CollectionDict Set a
collectionSetDict =
    ( \a s -> insert a s
    , \a -> insert a empty
    , \s -> if s == empty then Nothing else Just (deleteFindMin s)
    )
```

Functions whose argument types are constrained by type classes are then modified to take an additional dictionary argument, and references to type class functions are replaced with indexes into the dictionary:

```
getTwo :: CollectionDict c a -> c a -> Maybe (c a, a, a)
getTwo dict c = do (c' , a ) <- ((\(_, _, r) -> r) dict) c
                   (c'', a') <- ((\(_, _, r) -> r) dict) c'
                   return (c'', a, a')
```

This adds additional boilerplate of inserting the correct dictionary into each call to `getTwo`. With implicit functions, we can remove this boilerplate by making the dictionary parameter implicit, resulting in call sites being unchanged vs languages with type classes. We can also define functions that index into the dictionary as implicit functions, allowing us to use the original type class function names. We show this in the syntax of Haskell's `ImplicitParams` language extension [Lewis et al., 2000]:

```
add :: (?dict :: CollectionDict c a) => a -> c a -> c a
add = (\(a, _, _) -> a) ?dict

singleton :: (?dict :: CollectionDict c a) => a -> c a
singleton = (\(_, s, _) -> s) ?dict

remove :: (?dict :: CollectionDict c a) => c a -> Maybe (c a, a)
remove = (\(_, _, r) -> r) ?dict
```

Implicits are conceptually simpler than type classes, and therefore easier to implement, however they do require slightly more boilerplate code compared to type classes in order to achieve ad-hoc polymorphism. There is an argument to be made, however, that the minimal boilerplate reduction of type classes over implicits does not justify the additional complexity in the compiler.

It was recently announced [Odersky, 2019] that Scala 3 will include a new feature called *delegates*, which are essentially type classes, and are implemented reusing the pre-existing implicit resolution logic in the Scala compiler. Similarly, Haskell's `ImplicitParams` extension leverages the compiler's type class resolution logic to implement implicit parameters. These two facts testify to the relatedness of the two concepts.

2.3.7 Linearity

The *Linear Lambda Calculus* (LLC), inspired by ideas in linear logic [Girard, 1987], is a *substructurally typed* lambda calculus that places restrictions on the usage of variables. In LLC, each variable must be used exactly once. This property allows LLC to ascertain useful properties of programs with the type system, such as whether all opened files are closed, whether memory is double-deallocated or whether it is not deallocated at all. LLC introduces linear function types $T \multimap T'$ to represent functions whose bound variable is linear, i.e. used exactly once. More precisely, for linear functions $\lambda x.M$, the body M must contain exactly one occurrence of x. Formulations of LLC often include standard, or *unrestricted* functions $T \to T'$ alongside linear ones, distinguishing them at the term level with *qualifiers* `lin` and `un`, i.e. `lin` $\lambda x.M$ for linear functions and `un` $\lambda x.M$ for unrestricted ones. This improves their usability by allowing the expression of typical (nonlinear) programs whilst allowing linearity guarantees where desirable[6].

Control of aforementioned properties such as file closure is achieved in LLC using linear restriction of file variables. We might expect a file API to have the following function types [Walker, 2005]:

[6]Girard's linear logic [Girard, 1987] originally used *modalities* for this.

```
openFile :: String ⊸ File
readLine :: File ⊸ (File, un String)
closeFile :: File ⊸ ()
```

`openFile` takes a `String` file name and returns a linear `File` (note types are linear unless marked with `un`). Since the `File` is linear, we would expect to only be able to use it once, which would be problematic - we could only read a single line from it, leaving it open after execution completes, or close it immediately without reading from it. Instead, the `readLine` function returns a pair containing another `File`, which will reference the same file, but as a fresh linear variable that can be used again (once), along with an unrestricted `String` containing the read line. In this manner, each time we read a line, we get another linear reference to the `File` to read from again or close, but when we close the `File` with the function `closeFile`, no `File` reference is returned, and therefore the `File` cannot be used any further. Any attempts to use old references to the `File` will cause the type check to fail as this would result in the usage of a linear variable more than once. If we fail to close the file, at some point there must have been an unused linear variable, which LLC does not allow.

Figure 2.10 gives the syntax of LLC [Walker, 2005]. Qualifiers `q` allow the restriction of functions, tuples and types.

M, N	::=	$\mathtt{q}\ \lambda x.M$	(ABSTRACTION)
	|	$M\ N$	(APPLICATION)
	|	x	(VARIABLE)
	|	$\mathtt{q}\ (M, N)$	(TUPLE)
	|	$\mathtt{let}\ x = M\ \mathtt{in}\ N$	(LET-BINDING)
	|	$\mathtt{let}\ x, y = M\ \mathtt{in}\ N$	(TUPLE BINDING)
x	::=	(*variable names*)	
$\mathtt{q}$	::=	$\mathtt{lin} \mid \mathtt{un}$	(QUALIFIERS)
T	::=	$\mathtt{q}\ P$	(QUALIFIED PRETYPE)
P	::=	$T \to T$	(FUNCTION)
	|	$T \multimap T$	(LINEAR FUNCTION)
	|	(T, T)	(TUPLE)

FIGURE 2.10: Syntax of Linear Lambda Calculus

Semantics are unchanged from STLC after erasure of qualifiers `q`. Standard typing rules for systems such as STLC can freely *copy* contexts, for example when typing an application, the same context can be used to type the function expression and the argument expression. With linear systems, freely allowing copying would break linearity constraints, as we could freely use a linear variable in both the function expression and the argument expression. For this reason we have *context splitting rules* which allow us

to *split* contexts Γ. The context splitting rules allow us to freely copy unrestricted variables (see [UNSPLIT]), but do not allow us to copy linear variables - when we split a context, each linear variable may occur in only one of the obtained contexts, being usable in typing only the argument expression and not the function expression, or vice-versa (see [LINSPLITL, LINSPLITR]). Context splitting rules for LLC are given in figure 2.11 and typing rules in figure 2.12.

$$\frac{-}{\emptyset = \emptyset \circ \emptyset}\ [\textsc{Empty}] \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathtt{lin}\ P = (\Gamma_1, x : \mathtt{lin}\ P) \circ \Gamma_2}\ [\textsc{LinSplitL}]$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathtt{lin}\ P = \Gamma_1 \circ (\Gamma_2, x : \mathtt{lin}\ P)}\ [\textsc{LinSplitR}]$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathtt{un}\ P = (\Gamma_1, x : \mathtt{un}\ P) \circ (\Gamma_2, x : \mathtt{un}\ P)}\ [\textsc{UnSplit}]$$

FIGURE 2.11: Context Splitting Rules for Linear Lambda Calculus

$$\frac{\Gamma_1 \vdash M : \mathtt{q}\ T \to U \quad \Gamma_2 \vdash N : T}{\Gamma_1 \circ \Gamma_2 \vdash M\ N : U}\ [\textsc{App}] \qquad \frac{\mathtt{q}(\Gamma) \quad \Gamma, x : T \vdash M : U}{\Gamma \vdash \mathtt{q}\ \lambda x.M : \mathtt{q}\ T \to U}\ [\textsc{Abs}]$$

$$\frac{\mathtt{un}(\Gamma_1) \quad \mathtt{un}(\Gamma_2)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T}\ [\textsc{Var}] \qquad \frac{\mathtt{q}(T) \quad \mathtt{q}(U) \quad \Gamma_1 \vdash M : T \quad \Gamma_2 \vdash N : U}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{q}\ (M, N) : \mathtt{q}\ (T, U)}\ [\textsc{Tuple}]$$

$$\frac{\Gamma_1 \vdash M : \mathtt{q}\ T \quad \Gamma_2, x : T \vdash N : U}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N : U}\ [\textsc{Let}]$$

$$\frac{\Gamma_1 \vdash M : \mathtt{q}\ (T, U) \quad \Gamma_2, x : T, y : U \vdash N : V}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{let}\ x, y = M\ \mathtt{in}\ N : V}\ [\textsc{LetTup}]$$

FIGURE 2.12: Typing Rules for Linear Lambda Calculus

The typing rules [APP] and [LET] type application and let-binding respectively in the usual way, with the addition that the context is split as described to prevent violation of linearity constraints. For the same reason, the rule [VAR] must check that the context not consumed is unrestricted. The rules [TUP] and [LETTUP] type tuple introduction and elimination respectively in the standard way, again with context splitting.

2.4 Concurrency

We now leave the world of sequential computation to introduce the field of concurrent computation. So far all the languages we've seen carry out a single operation at a time,

in a fixed sequence. Concurrency concerns forms of computation in which the order of operations are not restricted to a fixed sequence, but progress of operations may be interleaved or simultaneous. We will look at early models of concurrency such as CCS, and more modern developments such as Pi Calculus and Session Types.

2.4.1 Terminology

The term '*concurrency*' is used to describe kinds of computational phenomena that can be thought of as having many things happen simultaneously. The term is often used interchangeably with the terms '*parallel*' and '*distributed*', although they are not synonyms. Concurrency is the most general of these three terms, implying the least about the computation or process(es) it describes. The term 'parallel' is usually understood as referring to processes that actually progress simultaneously, whereas 'concurrency' can describe a system where processes are 'interleaved', each making progress before the other finishes but never actually progressing simultaneously, such as in a multitasking operating system on a single-processor computer. The term 'distributed' refers to those parallel phenomena in which processes are in some sense disparate, such as two computers communicating over the internet.

With this in mind, we might say that all distributed phenomena are parallel, and all parallel phenomena are concurrent, but not all concurrent phenomena are parallel, and not all parallel phenomena are distributed.

Communication is generally thought of as a fundamental part of the concepts of concurrency, parallelism and distribution. Two processes may not necessarily be thought of as concurrent unless they have some form of interaction with one another, such as exchanging messages, or observing one another's behaviour. Where two processes exhibit no kind of communication, we might not describe them as concurrent, but as entirely separate processes that happen to be temporally collocated.

2.4.2 Forms of Concurrency

Concurrency is everywhere - 8 planets concurrently orbit the sun, and simultaneously exert gravitational forces on one another. Bus services operate concurrently with train services, within which separate buses drive around concurrently. Computers, networks of computers and networks of networks of computers communicate concurrently in order to form the internet.

In computing, concurrency also occurs at many different levels. Multitasking operating systems manage many concurrently running processes. Networks of computers send messages to one another concurrently. In general, concurrency is implemented in one of two ways: *Message Passing* or *Shared Memory*.

Message Passing

In a message passing system, concurrent processes communicate with each other via the exchange of messages. A message passing system may have a notion of *sender* and *receiver*, such that one message sends, and another receives, a single message. A given process may be a sender at one point in time, and later change its behaviour and become a receiver. Such a system might also contain the notion of *broadcast* messages, where a single message can have multiple receivers. There also exists a notion of message *channels*, whereby each message is sent over a specific channel. Senders must send on the same channel on which a receiver listens, in order to communicate.

Shared Memory

In a shared memory system, processes communicate by reading to, and writing from, a region of shared memory. In this scheme, the notion of sender and receiver breaks down somewhat. Processes may read from and write to regions of memory, whilst others do the same. Communication is generally achieved by the use of a pre-agreed scheme, where, for example, process A reads from, and process B writes to memory region X, and A writes to, and B reads from Y.

Implementations of shared memory schemes are often complicated by *synchronisation* problems. Such problems are elegantly illustrated by the well-known *dining philosophers* scenario, described in [Hoare, 1985, p. 75].

2.4.3 Theoretical Models of Concurrency

There exist many theoretical models of concurrency. These theoretical models are idealised mathematical descriptions of concurrent computation, as lambda calculi are theoretical models of sequential computation. Theoretical models of concurrency are useful in allowing us to study concurrent phenomena in the abstract. Thus we need not deal with problems that only exist in real-world implementations, such as computers in a network going off-line. There exist many theoretical models of concurrent computation, such as CSP [Hoare, 1985], CCS [Milner, 1989], the actor model [Agha, 1986] and the pi calculus [Milner, 1999]. We will now look at two of these notions in detail.

2.4.4 Calculus of Communicating Systems (CCS)

One of the more interesting models of concurrent computation is CCS. CCS is a message passing model of concurrent computation. It has primitives that express the following behaviours:

- message passing – $\bar{x}(e).P$ denotes a process that sends a message e over the channel x, and then executes the process P. $x(y).P$ denotes a process that receives a message y on the channel x and then executes P, and free occurrences of y in P are bound to the received message
- *parallel composition* $P \mid Q$ – the process P executes in parallel with the process Q
- *nondeterministic choice* $P + Q$, which denotes a process that can behave either as P or as Q
- *name restriction* $P \setminus L$, which represents the process P restricted to external communication over channels not in the set L
- *relabelling* $P[f]$, which allows for communication over restricted channels by renaming them with the function f
- *recursive definitions* $X \stackrel{\text{def}}{=} P$, which defines the process variable X as representing the process P, allowing for recursion (e.g. $X \stackrel{\text{def}}{=} x(y).X$) and reuse of processes.

The language has a notion of channel *names* and *co-names*. Names represent the channels for communication. Where a process receiving on the channel α is composed in parallel with a process sending on the channel $\bar{\alpha}$ ($\bar{\alpha}$ is the co-name of α), communication can occur.

The syntax of CCS is given in figure 2.13 [Milner, 1989, p. 43]). Expressions e represent a simple expression language with variables and constants such as integers and addition - the precise details of the expression language is unimportant, except that they contain variables that can refer to received messages.

A concurrent system is modelled in CCS as a non-empty, finite list of recursive definitions, given by the D production. Labelled transition semantics for CCS are given in figure 2.14. [Milner, 1989, p. 46]:

The version of CCS presented here is referred to as *value-passing* CCS. In value-passing CCS, messages contain some sort of contents. The nature of these contents are somewhat arbitrary. We might define our message content language to be a language of arithmetic expressions, or the lambda calculus, for example.

In the presented version of CCS, message output actions send a message in the message content language, over a particular channel. The message content is evaluated within whatever message content language is being used, and the resulting expression parameterises a corresponding variable in the receiver of the message. For example, we might define the agent $ADDER$, that performs addition:

$$ADDER \stackrel{\text{def}}{=} a(x).a(y).\bar{a}(x+y).ADDER$$

$$
\begin{array}{lcll}
D & ::= & X \overset{\mathbf{def}}{=} P & \text{(RECURSIVE DEFINITION)} \\
P, Q & ::= & P \mid Q & \text{(PARALLEL COMPOSITION)} \\
 & | & P + Q & \text{(NONDETERMINISTIC CHOICE)} \\
 & | & P[f] & \text{(RELABELLING)} \\
 & | & P \setminus L & \text{(RESTRICTION)} \\
 & | & X & \text{(RECURSIVE CALL)} \\
 & | & x(v).P & \text{(MESSAGE INPUT)} \\
 & | & \bar{x}(e).P & \text{(MESSAGE OUTPUT)} \\
X & ::= & \text{(\textit{definition names})} & \\
[f] & ::= & \text{(\textit{list of unary functions over the set of channel names and co-names})} & \\
x & ::= & \text{(\textit{channel names})} & \\
\bar{x} & ::= & \text{(\textit{channel co-names})} & \\
e & ::= & \text{(\textit{expressions})} & \\
v & ::= & \text{(\textit{expression variables})} &
\end{array}
$$

FIGURE 2.13: Syntax of CCS

$$\frac{}{\alpha(v).P \overset{\bar{\alpha}(e)}{\rightarrow} P[e/v]} \quad [\text{ACTIN}] \qquad \frac{}{\bar{\alpha}(e).P \overset{\bar{\alpha}(e)}{\rightarrow} P} \quad [\text{ACTOUT}]$$

$$\frac{P \overset{\bar{\alpha}(e)}{\rightarrow} P'}{P + Q \overset{\bar{\alpha}(e)}{\rightarrow} P'} \quad [\text{SUML}] \qquad \frac{Q \overset{\bar{\alpha}(e)}{\rightarrow} Q'}{P + Q \overset{\bar{\alpha}(e)}{\rightarrow} Q'} \quad [\text{SUMR}]$$

$$\frac{P \overset{\bar{\alpha}(e)}{\rightarrow} P'}{X \overset{\bar{\alpha}(e)}{\rightarrow} P'} (X \overset{\mathbf{def}}{=} P) \quad [\text{DEF}] \qquad \frac{P \overset{\bar{\alpha}(e)}{\rightarrow} P'}{P \mid Q \overset{\bar{\alpha}(e)}{\rightarrow} P' \mid Q} \quad [\text{COM1}]$$

$$\frac{Q \overset{\bar{\alpha}(e)}{\rightarrow} Q'}{P \mid Q \overset{\bar{\alpha}(e)}{\rightarrow} P \mid Q'} \quad [\text{COM2}] \qquad \frac{P \overset{\bar{\alpha}(e)}{\rightarrow} P' \qquad Q \overset{\bar{\alpha}(e)}{\rightarrow} Q'}{P \mid Q \overset{\tau}{\rightarrow} P' \mid Q'} \quad [\text{COM3}]$$

$$\frac{P \overset{\bar{\alpha}(e)}{\rightarrow} P'}{P \setminus L \overset{\bar{\alpha}(e)}{\rightarrow} P' \setminus L} (\alpha, \bar{\alpha} \notin L) \quad [\text{RES}] \qquad \frac{P \overset{\bar{\alpha}(e)}{\rightarrow} P'}{P[f] \overset{f(\bar{\alpha}(e))}{\rightarrow} P'[f]} \quad [\text{REL}]$$

FIGURE 2.14: Semantics of CCS

When composed with an agent that makes use of *ADDER*, we see the how value passing works:

$$
\begin{array}{rrcl}
 & \bar{a}(1).\bar{a}(2).a(x).\bar{b}(x) & \mid & ADDER \\
\text{by [DEF]} & \bar{a}(1).\bar{a}(2).a(x).\bar{b}(x) & \mid & a(x).a(y).\bar{a}(x+y).ADDER \\
\text{by [COM3]} & \bar{a}(2).a(x).\bar{b}(x) & \mid & a(y).\bar{a}(1+y).ADDER \\
\text{by [COM3]} & a(x).\bar{b}(x) & \mid & \bar{a}(1+2).ADDER \\
\text{by [COM3]} & \bar{b}(3) & \mid & ADDER
\end{array}
$$

We can see that our agent sends messages 1 and 2 over a, to the $ADDER$ agent. The $ADDER$ agent then replies with their sum, 3.

We might also consider a version of CCS where we omit message content from agents altogether. Agents simply *synchronise* or *handshake* over channels, rather than pass messages.

A *silent action*, denoted above with τ, is an action that occurs inside a process. The simplest case of a silent action is an internal communication. Consider the process $P \mid Q$, such that: $P \stackrel{\text{def}}{=} a(v).P'$, and: $Q \stackrel{\text{def}}{=} \bar{a}(e).Q'$, then P and Q will communicate over channel a, and become P' and $Q'[e/v]$. If we consider P and Q individually, rule [COM1] applies to P and [COM2] applies to Q. When considering the process as a whole, for example in relation to a separate process, this action is captured by [COM3]. The outside observer sees the transition $P \mid Q \xrightarrow{\tau} P' \mid Q'[e/v]$. Here τ denotes the message exchanged between P and Q. The channel name a and content e are not seen.

2.4.5 Pi Calculus

Another useful model of concurrent computation is Milner's pi calculus [Milner, 1999]. The pi calculus has features similar to CCS, but with a key addition: mobility. In the pi calculus, the names given to channels are themselves the content of messages, as opposed to a separate expression language. This allows agents to dynamically change which other agents they communicate with. It also allows for the dynamic introduction of new agents.

The syntax of the pi calculus is given in figure 2.15 [Milner, 1999, p. 87].

$$
\begin{array}{lcll}
P, Q & ::= & P \mid Q & \text{(PARALLEL COMPOSITION)} \\
 & \mid & a(x).P & \text{(MESSAGE INPUT)} \\
 & \mid & \bar{a}x.P & \text{(MESSAGE OUTPUT)} \\
 & \mid & !P & \text{(RECURSION)} \\
 & \mid & 0 & \text{(EMPTY PROCESS)} \\
 & \mid & (\nu x)P & \text{(RESTRICTION)} \\
a, x & ::= & \textit{(channel names)} & \\
\bar{a}, \bar{x} & ::= & \textit{(channel co-names)} &
\end{array}
$$

FIGURE 2.15: Syntax of Pi Calculus

We cannot give the semantics of the pi calculus without first defining the notion of *structural congruence*. Structural congruence rules are laws that express behavioural identities between processes that are syntactically distinct. The structural congruence rules for the pi calculus are given in figure 2.16.

$$\begin{array}{rcl@{\qquad}rcl} P \mid 0 & \equiv & P & P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\ P \mid Q & \equiv & Q \mid P & (\nu x)(P \mid Q) & \equiv & ((\nu x)P) \mid Q \quad \textbf{if} \quad x \notin fn(P) \\ (\nu x)0 & \equiv & 0 & (\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P \\ !P & \equiv & P \mid !P & & & \end{array}$$

FIGURE 2.16: Structural congruence rules for Pi Calculus

In the structural congruence rules above, a reference is made to the *free names* function, fn. This function computes the set of free names within a process. The function is defined as follows:

$$\begin{array}{rcl@{\qquad}rcl} fn(P \mid Q) & = & fn(P) \cup fn(Q) & fn(0) & = & \{\} \\ fn(a(x).P) & = & fn(P) - x & fn(\bar{a}x.P) & = & fn(P) \cup \{x\} \\ fn(!P) & = & fn(P) & fn((\nu x)P) & = & fn(P) - x \end{array}$$

We have defined the syntax and structural congruence rules of the pi calculus, and also defined free names in the pi calculus. We can now give the semantics for the pi calculus. They are given in figure 2.17 as reduction semantics.

$$\frac{}{a(y).P \mid \bar{a}x.Q \quad \rightarrow \quad P[x/y] \mid Q} \quad [\text{REACT}] \qquad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \quad [\text{PAR}]$$

$$\frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'} \quad [\text{RES}] \qquad \frac{P \rightarrow P'}{Q \rightarrow Q'}(P \equiv Q, P' \equiv Q') \quad [\text{STRUCT}]$$

FIGURE 2.17: Semantics of Pi Calculus

Recursion

The pi calculus achieves recursion differently to CCS. The pi calculus uses a recursion combinator as opposed to recursive definitions. The recursion combinator !, applied to a process P, denotes in some sense the repetition of the process P. For example, consider a

process that repeatedly sends a message x on channel a. In CCS we have:

$$A \quad \texttt{where} \quad A \stackrel{\textbf{def}}{=} \bar{a}(x).A$$

In the pi calculus we would have:

$$!\bar{a}x.0$$

The pi calculus version of this process does behave in a subtly different way to the CCS - due to the structural congruence $!P \equiv P \mid !P$, multiple processes can make use of the process $!\bar{a}x.0$ simultaneously. Consider the example:

$$a(y).\tau.P \mid a(z).\tau.Q \mid !\bar{a}x.0$$

Due to the congruence $!P \equiv P \mid !P$, this behaves as:

$$a(y).\tau.P \mid a(z).\tau.Q \mid \bar{a}x.0 \mid !\bar{a}x.0$$

Which can behave as:

$$a(y).\tau.P \mid a(z).\tau.Q \mid \bar{a}x.0 \mid \bar{a}x.0 \mid !\bar{a}x.0$$

Which allows the receiving processes to concurrently react with the separate copies of $\bar{a}x.0$, where in CCS the first receiving process would have to complete its communication with the recursive process before the second receiving process could communicate with the recursive process. In the pi calculus, we could have the reduction sequence:

$$(\tau.P)[x/y] \mid a(z).\tau.Q \mid \bar{a}x.0 \mid !\bar{a}x.0$$

$$(\tau.P)[x/y] \mid (\tau.Q)[x/z] \mid !\bar{a}x.0$$

$$P[x/y] \mid (\tau.Q)[x/z] \mid !\bar{a}x.0$$

$$P[x/y] \mid Q[x/z] \mid !\bar{a}x.0$$

With recursive definitions, the reaction of the first receiver must complete, yielding $P[x/y]$, before the process $a(y).\tau.Q$ can begin its reaction with the repeating sender. Thus, the behaviour in the above example is not (trivially) possible in CCS.

Mobility

As discussed, mobility is the term used to refer to the property of the pi calculus that allows dynamic reconfiguration of communication networks. The pi calculus makes mobility possible by allowing the sending of message channel names. An agent might receive a channel name as a message, and proceed to send a message over that channel. In

this way, agents dynamically change the other agents they can communicate with. Such behaviour is not possible in CCS.

2.4.6 Equality in models of computation

It is often important to define a notion of equality between programs, within a given model of computation. This is because a notion of program equality is required in order to have a notion of compiler correctness. Consider a compiler C, from language A to language B, and two programs in language A, a_1 and a_2. We say that such a compiler C is *sound* if whenever two compiled programs that are equal in language B ($b_1 \equiv_B b_2$), it follows that their source programs were also equal ($a_1 \equiv_A a_2$). So for all programs a_1 and a_2, if $b_1 \equiv_B C(a_1)$, $b_2 \equiv_B C(a_2)$ and $b_1 \equiv_B b_2$, imply that $a_1 \equiv_A a_2$, then the compiler C is sound.

Soundness is often phrased differently (but equivalently): A compiler from A to B is sound if a property P holding for a target program b, i.e. $P(b)$ implies that it must have held for the source program a, i.e. $P(a)$, so $P(b) \Rightarrow P(a)$.

Without a formal notion of program equality, we cannot formalise a notion of compiler correctness (soundness). Thus, if we are to show that a compiler for the pi calculus is correct, we must have a formal definition of process equality for the pi calculus, and a definition for whatever language we compile it from or to.

There are other notions of compiler correctness such as *completeness* and *full abstraction* (soundness *and* completeness). Completeness is the dual of soundness: $b_1 \equiv_B C(a_1) \wedge b_2 \equiv_B C(a_2) \wedge a_1 \equiv_A a_2 \Rightarrow b_1 \equiv_B b_2$ or equivalently $P(a) \Rightarrow P(b)$. Notions other than soundness are, however, not generally required for practical programming languages, except in the cases where security is of utmost concern.

Notions of equality for concurrent programming languages are often more complicated than for sequential ones, since concurrency allows programs to *observe* other programs as they execute. Sequential programs may only compare inputs and outputs. It is not always clear what an appropriate notion of equality is for a given concurrent language. This problem must be solved before soundness can be shown.

2.4.7 Implementing the Pi Calculus

The pi calculus can be implemented via an abstract machine [Turner, 1995, p. 99]. This machine uses two data structures: a queue of runnable processes (here referred to as the *run queue*), and a mapping from channel names to queues of processes that wait to communicate on the channel that maps to them (here referred to as the *wait map*). Specifically, each channel in the wait map maps to a single queue, containing *either* only processes waiting to send *or* only processes waiting to receive. This queue does not mix senders or receivers (it need not, since if there are processes waiting to send on a given channel,

and processes waiting to receive on that channel, they may communicate and need not be waiting at all).

We define update rules which modify the state of the abstract machine, thereby executing the process. Each update rule advances the state of the abstract machine by one *step*. The update rules are repeatedly applied until execution finishes.

To run a pi calculus term on the abstract machine, we initialise the wait map to be empty, and the run queue to contain only the term we want to run, and then update the machine according to the update rules, until the run queue is empty. The processes left in the wait map once the run queue is empty are those that cannot be reduced, and can be seen as the result of the execution.

To simplify the abstract machine, we use a simplified pi calculus, where instead of having general process replication, replication is restricted to receiving processes. We use a calculus with the grammar:

$$P, Q ::= \quad P \mid Q \mid (\nu x)P \mid x(a).P \mid x!(a).P \mid \bar{x}a.P \mid 0$$

where $x!(a).P$ denotes a replicated receiving process, that behaves as the process $!x(a).P$ in the full calculus. We do not lose expressive power by making this simplification, since full pi calculus processes can be compiled into the simplified calculus, with the function *transRR*, defined below. This function takes a full pi calculus term and maps it to a simplified pi calculus term with the same behaviour.

$$\begin{aligned}
transRR(\bar{x}a.P) &= \bar{x}a.transRR(P) \\
transRR(x(a).P) &= x(a).transRR(P) \\
transRR((\nu x)P) &= (\nu x)transRR(P) \\
transRR(P \mid Q) &= transRR(P) \mid transRR(Q) \\
transRR(0) &= 0 \\
transRR(!x(a).P) &= x!(a).transRR(P) \\
transRR(!P) &= (\nu x)(\nu y)(x!(y).\bar{x}y.transRR(P) \mid \bar{x}y.0) \quad \texttt{where} \quad x, y \notin fn(P)
\end{aligned}$$

The reduction rules for the abstract machine are defined as a function, *step*, given in figure 2.18. Each rule takes the current run queue and wait map (denoted rq and wm respectively), and returns an updated run queue and wait map. The *step* function serves as an interpreter if executed repeatedly until $rq = \bullet$ (the empty list).

Other implementations of the pi calculus are possible. For example, [Turner, 1995, p. 125] presents an implementation of the pi calculus via compilation to C.

$$
\begin{aligned}
&step(wm, \bullet) = wm, rq \\
&step(wm, T :: rq) = \texttt{case } T \texttt{ of} \\
&\qquad 0 \rightarrow wm, rq \\
&\qquad P \mid Q \rightarrow wm, P :: rq :: Q \\
&\qquad (\nu x)P \rightarrow \\
&\qquad\quad \texttt{let } c = freshname() \texttt{ in} \\
&\qquad\quad wm :: (c \rightarrow \bullet), P[c/x] :: rq \\
&\qquad c(x).P \texttt{ if } lookup(wm, c) = \bar{c}y.Q :: moresends \rightarrow \\
&\qquad\quad wm :: (c \rightarrow moresends), P[y/x] :: rq :: Q \\
&\qquad c(x).P \texttt{ if } lookup(wm, c) = morerecs \rightarrow \\
&\qquad\quad wm :: (c \rightarrow morerecs :: c(x).P), rq \\
&\qquad \bar{c}x.P \texttt{ if } lookup(wm, c) = c(y).Q :: morerecs \rightarrow \\
&\qquad\quad wm :: (c \rightarrow morerecs), P :: rq :: Q[x/y] \\
&\qquad \bar{c}x.P \texttt{ if } lookup(wm, c) = c!(y).Q :: morerecs \rightarrow \\
&\qquad\quad wm :: (c \rightarrow morerecs :: c!(y).Q), P :: rq :: Q[x/y] \\
&\qquad \bar{c}x.P \texttt{ if } lookup(wm, c) = moresends \rightarrow \\
&\qquad\quad wm :: (c \rightarrow moresends :: \bar{c}x.P), rq \\
&\qquad c!(x).P \texttt{ if } lookup(wm, c) = \bar{c}y.Q :: moresends \rightarrow \\
&\qquad\quad wm :: (c \rightarrow moresends), c!(x).P :: rq :: P[y/x] :: Q \\
&\qquad c!(x).P \texttt{ if } lookup(wm, c) = morerecs \rightarrow \\
&\qquad\quad wm :: (c \rightarrow morerecs :: c!(x).P), rq
\end{aligned}
$$

FIGURE 2.18: Abstract Machine for Pi Calculus

Nondeterminsim

A typical property of process calculi is that they tend to contain aspects of nondeterministic behaviour. In the case of CCS, we even have a nondeterministic choice operator $P + Q$. The grammar and semantics of the pi calculus are often extended to include such an operator. Not only is nondeterministic behaviour captured by nondeterministic choice operators, but also in the very nature of message passing. We do not impose any time or order constraints on the flow of messages. Consider the following pi calculus term:

$$a(w).P \mid a(x).Q \mid \bar{a}y.R \mid \bar{a}z.S$$

Below are all four terms that can be reached from this term in a single reduction step:

$$P[y/w] \mid a(x).Q \mid R \mid \bar{a}z.S$$

$$a(w).P \mid Q[y/x] \mid R \mid \bar{a}z.S$$

$$P[z/w] \mid a(x).Q \mid \bar{a}y.R \mid S$$

$$a(w).P \mid Q[z/x] \mid \bar{a}y.R \mid S$$

There are two possible terms once all possible reactions have occurred (assuming P, Q, R, S do not react):

$$P[y/w] \mid Q[z/x] \mid R \mid S$$

$$P[z/w] \mid Q[y/x] \mid R \mid S$$

So not only can we take multiple execution paths to reach a single reduced form, but multiple reduced forms are possible. This is therefore nondeterministic behaviour. In fact, we can use the nondeterministic message passing to simulate nondeterministic choice. We demonstrate this with a translation function *transND*, from a pi calculus with a nondeterministic choice operator, into the basic pi calculus presented earlier. The translation resembles those given in [Nestmann and Pierce, 1996]. Our calculus with a nondeterministic choice operator has the grammar:

$$P, Q ::= \quad P|Q \mid (\nu x)P \mid x(a).P \mid x!(a).P \mid \bar{x}a.P \mid 0 \mid P + Q$$

We define *transND* as follows:

$$\begin{aligned}
transND(\bar{x}a.P) &= \bar{x}a.transND(P) \\
transND(x(a).P) &= x(a).transND(P) \\
transND((\nu x)P) &= (\nu x)transND(P) \\
transND(P \mid Q) &= transND(P) \mid transND(Q) \\
transND(0) &= 0 \\
transND(!P) &= !transND(P) \\
transND(P + Q) &= (\nu x)(\nu y)(\bar{x}y.P \mid \bar{x}y.Q \mid x(y).0) \quad \texttt{where} \quad x, y \notin fn(P) \cup fn(Q)
\end{aligned}$$

The function *transND* demonstrates that adding a nondeterministic choice operator to the pi calculus adds no expressive power.

Any implementation of the pi calculus must implement nondeterminism. True nondeterminism is not strictly possible in a real implementation and must be emulated. This is because modern computing devices do not exhibit any nondeterministic behaviour. The abstract machine resolves nondeterminism by choosing which processes react based on the order in which they appear. There are other ways of implementing nondeterminism, such as with probabilistic random choice. All possible reactions between processes can be computed, and one selected probabilistically. It stands to reason that such an implementation must in some sense be fair to all possible reactions, in the way that it assigns probabilities to possibilities. If it does not do so, there is a risk of processes being starved if they have a comparatively low probability of execution. It should be noted that probabilistic choice is not the same as nondeterminism - probabilistic choice assigns probabilities to all possibilities, whereas nondeterminism is a more abstract concept, specifying only what is *possible*, and does not specify what is *probable*. The pi calculus is not probabilistic, and so any probabilistic implementation must make its own choice about how reactions are selected.

Encoding the Lambda Calculus in the Pi Calculus

An approach to demonstrating the Turing-universality of pi calculus is to find an encoding of a Turing-universal model of computation in pi calculus. This was first achieved in [Milner, 1992] for lambda calculus. Encodings of both call-by-name and call-by-value lambda calculus are possible, and have been shown to be sound. The encodings are given in figure 2.19.

CALL-BY-NAME

$$
\begin{aligned}
[\![\lambda x.M]\!]_u &\stackrel{\text{def}}{=} u(x).u(v).[\![M]\!]_v \\
[\![x]\!]_u &\stackrel{\text{def}}{=} \overline{x}u \\
[\![M\ N]\!]_u &\stackrel{\text{def}}{=} (\nu v)([\![M]\!]_v | (\nu x)\overline{v}x.\overline{v}u.!x(w).[\![N]\!]_w) \quad (x \notin fn(N))
\end{aligned}
$$

CALL-BY-VALUE

$$
\begin{aligned}
[\![\lambda x.M]\!]_u &\stackrel{\text{def}}{=} (\nu y)\overline{u}y.!y(w).w(x).w(u).[\![M]\!]_u \quad (y \notin fn(\lambda x.M)) \\
[\![x]\!]_u &\stackrel{\text{def}}{=} (\nu y)\overline{u}y.!y(w).\overline{x}w \\
[\![M\ N]\!]_u &\stackrel{\text{def}}{=} (\nu q)(\nu r)(q(y).(\nu v)\overline{y}v.r(z).\overline{v}z.\overline{v}u | [\![M]\!]_q | [\![N]\!]_r)
\end{aligned}
$$

FIGURE 2.19: Encodings of the Lambda Calculus in the Pi Calculus

These translations are parameterised by a channel u. This channel is the means by which a process can interact with a translated lambda term when parallel-composed with it.

When stating that the translations are sound, we mean that for a notion of equality for pi terms $\equiv_\pi$, for a lambda term l, if $l \to l'$, then $[\![l]\!]_u \to p$ such that $p \equiv_\pi [\![l']\!]_u$. For these translations, our notion of equality for pi terms is essentially structural congruence with additional intuitive equivalence axioms, such as the law that captures the fact that processes such as $(\nu x)\overline{x}y.P$ and $(\nu x)x(y).P$ can never interact with any process, so parallel-composing another process with such a process is identity, so, for example, $Q|(\nu x)x(y).P$ is equivalent to Q.

We can demonstrate intuitively the soundness of the translations by example. We start with the lambda term $(\lambda x.x)y$, which we expect to reduce to just y. Soundness then dictates that $[\![(\lambda x.x)y]\!]_u$ is equivalent to $[\![y]\!]_u$. We now verify this, using the call-by-name translation:

- $[\![(\lambda x.x)y]\!]_u$ gives $(\nu v)(v(x).v(t).\overline{x}t|(\nu z)\overline{v}z.\overline{v}u.!z(w).\overline{y}w)$.
- We use structural congruence to bring the (νz) outwards, which is allowed since z is not free in $v(x).v(t).\overline{x}t$. We obtain $(\nu v)(\nu z)(v(x).v(t).\overline{x}t|\overline{v}z.\overline{v}u.!z(w).\overline{y}w)$.
- This reacts over v, forming $(\nu v)(\nu z)(v(t).\overline{z}t|\overline{v}u.!z(w).\overline{y}w)$.
- This reacts over v again, forming $(\nu v)(\nu z)(\overline{z}u|!z(w).\overline{y}w)$
- This reacts over z, forming $(\nu v)(\nu z)(!z(w).\overline{y}w|\overline{y}u)$
- We can now drop the (νv) since v does not occur in the body and therefore the (νv) cannot affect the term's behaviour. We obtain $(\nu z)(!z(w).\overline{y}w|\overline{y}u)$.
- We use structural congruence to bring the $\overline{y}u$ outside the (νz), which is allowed since z is not free in $\overline{y}u$. We obtain $(\nu z)!z(w).\overline{y}w|\overline{y}u$.
- We can now drop the $(\nu z)!z(w).\overline{y}w$, since the (νz) guard prevents the body from interacting in any way. We obtain $\overline{y}u$.
- We can now see that the obtained process, $\overline{y}u$, is what would be obtained from the translation $[\![y]\!]_u$, satisfying soundness as described above.

It is clear from the above that with this translation, what is a single reduction step in the lambda calculus becomes several steps in the pi calculus. Type systems can be used with both the pi and lambda calculi to mitigate this discrepancy, and even to allow for a converse translation from pi to lambda [Toninho and Yoshida, 2017].

2.4.8 Types for Concurrency

Just as type systems find many applications in functional programming, with typed lambda calculi as their theoretical bases, many useful type systems also exist for concurrent programming languages, with typed pi calculi as their theoretical bases. Type

systems for concurrency can guarantee domain-relevant properties such as agreement between communication participants about the format of data being sent, agreed patterns of communication and deadlock freedom. We now look at some well-known typed concurrent programming languages.

Polymorphic channel types for Pi Calculus

A useful notion of types for pi calculus, drawing from polymorphic function types in the sequential world, are polymorphic channel types [Turner, 1995]. Polymorphic channel types impose strict typing on channels - each channel may carry values of a single type only.

$$
\begin{array}{rcll}
P & ::= & P \mid P & \text{(PARALLEL COMPOSITION)} \\
 & \mid & (\nu x : \delta)P & \text{(RESTRICTION)} \\
 & \mid & x?[\overline{\alpha}; \overline{y : \delta}].P & \text{(INPUT)} \\
 & \mid & x![\overline{\alpha}; \overline{y}].P & \text{(OUTPUT)} \\
 & \mid & *P & \text{(REPLICATION)} \\
 & \mid & \mathbf{0} & \text{(INACTION)} \\
 & & & \\
\delta & ::= & \uparrow[\overline{\alpha}; \overline{\delta}] & \text{(POLYMORPHIC CHANNELS)} \\
 & \mid & \alpha & \text{(TYPE VARIABLES)}
\end{array}
$$

FIGURE 2.20: Syntax of polymorphic pi calculus

The syntax of polymorphic pi calculus is given in figure 2.20. Here we denote replication with $*$ as opposed to !, and use ! to represent output, with ? denoting input, rather than using $\overline{\text{overline}}$ to distinguish input and output. We instead use $\overline{\text{overline}}$ to denote vectors. We annotate restriction with the type of values that will be exchanged over the restricted channel, and annotate input value binders with types also.

Channel types $\uparrow[\overline{\alpha}; \overline{\delta}]$ represent channels along which values of types given in the vector $\overline{\delta}$ are sent, and those types $\overline{\delta}$ may contain type variables found in the vector $\overline{\alpha}$. Typing rules for polymorphic pi calculus are given in figure 2.21. Rules [INACT], [PAR] and [REPL] are straightforward. The rule [RES] behaves similarly to rules for abstraction in lambda calculi, but instead of introducing a new variable, a new channel is introduced into the scope P - its type is added to the environment accordingly. The rules [INPUT] and [OUTPUT] type their namesake's communication operation. In the case of input, the bound variables types are checked to match the types declared for the channel in the environment. For output, the sent values are required to match in the same way. In this way it is enforced that channels only ever carry values of a single (parametrically polymorphic) type. If we add lists to the grammar of messages, and corresponding types,

$$\frac{-}{\Delta \vdash \mathbf{0}} \quad [\textsc{Inact}] \qquad \frac{\Delta \vdash P \quad \Delta \vdash Q}{\Delta \vdash P \mid Q} \quad [\textsc{Par}]$$

$$\frac{\Delta \vdash P}{\Delta \vdash *P} \quad [\textsc{Repl}] \qquad \frac{\Delta, x :\uparrow[\overline{\alpha};\overline{\delta}] \vdash P}{\Delta \vdash (\nu x :\uparrow[\overline{\alpha};\overline{\delta}])P} \quad [\textsc{Res}]$$

$$\frac{\begin{array}{c}\alpha_1, ..., \alpha_m \notin ftv(\Delta) \\ \Delta(c) =\uparrow[\alpha_1, ..., \alpha_m; \delta_1, ..., \delta_m] \quad \Delta, x_1 : \delta_1, ..., x_n : \delta_n \vdash P\end{array}}{\Delta \vdash c?[\alpha_1, ..., \alpha_m; x_1 : \delta_1, ..., x_n : \delta_n].P} \quad [\textsc{Input}]$$

$$\frac{\begin{array}{c}\Delta(c) =\uparrow[\alpha_1, ..., \alpha_m; \gamma_1, ..., \gamma_m] \\ \Delta(a_i) = [\delta_1,...,\delta_m/\alpha_1,...,\alpha_m]\gamma_i \quad 1 \leq i \leq n \quad \Delta \vdash P\end{array}}{\Delta \vdash c![\alpha_1, ..., \alpha_m; a_1, ..., a_n].P} \quad [\textsc{Output}]$$

FIGURE 2.21: Typing rules for polymorphic pi calculus

we can implement generic processes much like the generic functions over lists that are enabled by parametric polymorphism in HDMP.

Semantics for the polymorphic pi calculus match that of untyped polyadic pi calculus upon erasure of types. The usual structural congruences hold.

Polymorphic pi calculus corresponds to polymorphic lambda calculus (System F) [Reynolds, 1974; Girard, 1972]. A sound translation from the former to the latter is given in [Turner, 1995].

Linear types for Pi Calculus

Linear types are applicable and useful in the context of pi calculus as well as lambda calculus. In addition guaranteeing patterns of variable usage as in lambda calculus, linear types for pi calculus provide additional benefits such as optimisation of function-like communication patterns, and refinements to notions of process equivalence [Kobayashi, Pierce, and Turner, 1999]. Channels are controlled linearly in the linear pi calculus, meaning that communication on any one channel occurs exactly once. Therefore each channel variable must occur *twice*, rather than once as with LLC, since both the sender and receiver of a communication must refer to the channel for communication to occur.

Consider the process below:

$$plustwo? * [j, s].(\nu r_1)(\nu r_2)(plusone![j, r_1] \mid r_1?[k].plusone![k, r_2] \mid r_2?[l].s![l])$$

plustwo is a function-like process, that when sent a value j and return channel s, delivers the result $j + 2$ to the caller on the return channel s, much like the function $\lambda j.j + 2$. A problem with this process is that while the channel *plustwo* is used by the example to

provide a service, without linear channels, nothing prevents another process from also reading from the channel *plustwo*, but providing a different service, such as addition of 3. A type system that can prevent such a process from offering another service on *plustwo* is clearly desirable. The type system of linear pi calculus can prevent this.

Figure 2.22 gives the syntax of linear pi calculus. This presentation uses *asynchronous* output, where there is no process continuation after an output. Processes with continuations following an output such as $x?[].y![].z?[].\mathbf{0}$ are instead written $x?[].(y![]|z?[].\mathbf{0})$, meaning that the input on z can proceed immediately after the input on x, without waiting for a receiver to match the output on y. Channel types are annotated with *polarities* p and qualifiers $\mathtt{q}$. Qualifiers $\mathtt{q}$ have the same meaning as with LLC (§2.3.7) - in LLC a variable can be used once if its type is qualified with `lin`, and any number of times with `un`. The same holds for channel usage in linear pi calculus. Polarities p indicate how a channel is communicated over, i.e. what kind of endpoint it is. $\updownarrow$ indicates that a channel can be used for both input and output, ? indicates input only, ! indicates output only and | indicates that the channel may not be communicated over. Polarities can be thought of as sets of capabilities. Where i denotes input and o denotes output, $\updownarrow$ denotes the set $\{i, o\}$, ? the set $\{i\}$, ! the set $\{o\}$ and | the set $\{\}$. We take set operations on polarities p, q to be operations on these denoted sets.

The above example could be typed with the channel *plustwo* having the type $?^{\mathtt{lin}}[\mathtt{int}, !^{\mathtt{lin}}[\mathtt{int}]]$. Since the input capability ? is consumed by the example, it cannot typably be placed in parallel with another process that also reads from the *plustwo* channel, and thus the described error is prevented.

TERMS

P	::=	$P \mid P$	(PARALLEL COMPOSITION)
	|	$x![\overline{y}]$	(ASYNCHRONOUS OUTPUT)
	|	$x?[\overline{y}].P$	(INPUT)
	|	$x? * [\overline{y}].P$	(REPLICATED INPUT)
	|	$(\nu x : \delta)P$	(RESTRICTION)
	|	$\mathtt{if}\ x\ \mathtt{then}\ P\ \mathtt{else}\ P$	(CONDITIONAL)
	|	$\mathbf{0}$	(INACTION)

TYPES

δ	::=	$p^{\mathtt{q}}[\overline{\delta}]$	(CHANNEL TYPES)
	|	$\mathtt{Bool}$	(BOOLEANS)

QUALIFIERS

$\mathtt{q}$ ::= $\mathtt{lin} \mid \mathtt{un}$

POLARITIES

p	::=	$\updownarrow$	(INPUT AND OUTPUT)
	|	?	(INPUT ONLY)
	|	!	(OUTPUT ONLY)
	|	|	(NONE)

FIGURE 2.22: Syntax of linear pi calculus

Figure 2.24 gives the type combination rules for linear pi calculus. These fulfil the

$$\frac{\mathtt{un}(\Delta)}{\Delta \vdash \mathbf{0}} \quad [\textsc{Inact}] \qquad \frac{\mathtt{un}(\Delta)}{\Delta + x : !^{q}[\overline{\delta}] + \overline{y} : \overline{\delta} \vdash x![\overline{y}]} \quad [\textsc{Output}]$$

$$\frac{\Delta_1 \vdash P \quad \Delta_2 \vdash Q}{\Delta_1 + \Delta_2 \vdash P \mid Q} \quad [\textsc{Par}] \qquad \frac{\Delta, \overline{z} : \overline{\delta} \vdash P}{\Delta + x : ?^{q}[\overline{\delta}] \vdash x?[\overline{z}].P} \quad [\textsc{Input}]$$

$$\frac{\Delta, \overline{z} : \overline{\delta} \vdash P \quad \mathtt{un}(\Delta)}{\Delta + x : ?^{\mathtt{un}}[\overline{\delta}] \vdash x? * [\overline{z}].P} \quad [\textsc{RepInput}] \qquad \frac{\Delta, x : p^{q}[\overline{\delta}] \vdash P \quad p \in \{|, \updownarrow\}}{\Delta \vdash (\nu x : p^{q}[\overline{\delta}])P} \quad [\textsc{Res}]$$

$$\frac{\Delta \vdash P_1 \quad \Delta \vdash P_2}{\Delta + b : \mathtt{Bool} \vdash \mathtt{if}\ b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2} \quad [\textsc{If}]$$

FIGURE 2.23: Typing rules for linear pi calculus

$$\begin{aligned} \mathtt{Bool} + \mathtt{Bool} &= \mathtt{Bool} \\ p^{\mathtt{un}}[\overline{\delta}] + q^{\mathtt{un}}[\overline{\delta}] &= (p \cup q)^{\mathtt{un}}[\overline{\delta}] \\ p^{\mathtt{lin}}[\overline{\delta}] + q^{\mathtt{lin}}[\overline{\delta}] &= (p \cup q)^{\mathtt{lin}}[\overline{\delta}] \text{ if } p \cap q = \emptyset \end{aligned}$$

$$(\Delta_1 + \Delta_2)(x) = \begin{cases} \Delta_1(x) + \Delta_2(x) & \text{if } x \in dom(\Delta_1) \cap dom(\Delta_2) \\ \Delta_1(x) & \text{if } x \in dom(\Delta_1) \text{ and } x \notin dom(\Delta_2) \\ \Delta_2(x) & \text{if } x \in dom(\Delta_2) \text{ and } x \notin dom(\Delta_1) \end{cases}$$

FIGURE 2.24: Type combination for linear pi calculus

same function as the context addition rules for LLC. Unrestricted types are freely combined with like types, and in the case of channels this results in the union of their polarities. For linear channels, combination is possible only when the intersection of the argument's capability sets is empty, meaning that capabilities for one channel can never be common to multiple parties. This guarantees that for any linear channel, there is only ever one input endpoint and one output endpoint, and thus is used exactly once. Combination is extended to environments in a linearity-preserving manner as with LLC.

Figure 2.23 gives the typing rules for linear pi calculus. [INACT] types a syntax tree leaf as does [VAR] in LLC, and therefore the environment for typing must be unrestricted for the same reason. [PAR] types parallel composition but combines environments for each side of the composition like [TUP] for LLC. [INPUT] and [OUTPUT] are linearity-preserving and otherwise standard. [REPINPUT] allows for unrestricted channels only, since replication by definition allows many uses of the replicated process. [IF] preserves linearity by ensuring that both branches consume all linear channels, which appeases linearity since only one branch is executed. [RES] is only allowed on channels with polarity $\updownarrow$ or $|$, since restricting a channel that can only input or only output will prevent any communication and therefore violate linearity.

Semantics for the linear pi calculus are somewhat complicated over standard pi calculus due to linearity. The semantic rules require a context Δ with type information, as with typing. We do not show the full rules here, but the essence of the difference between them and standard labelled semantics is that we must check that we do not allow multiple reactions over linear channels - equally that a reaction over a linear channel x results in the polarity of x becoming $|$, and such a channel may not facilitate reaction. The calculus enjoys typical soundness properties such as type preservation under reduction and barbed congruence.

Session Types

Session types, first introduced in [Honda, 1993; Takeuchi, Honda, and Kubo, 1994], are an important example of a behavioural type system for message passing concurrency. Session types classify message passing behaviour at given channels: e.g. if process P first sends an integer on channel x, then receives a boolean on x, and finally sends a boolean on x, then this behaviour could be summarised by the session type

P : `!Int.?Bool.!Bool.end`

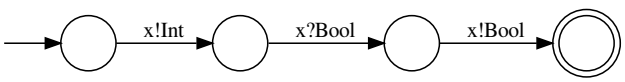
Here `?T` represents input of a value of type `T`, `!T` means sending a value that has type `T`, while `end` denotes the termination of the interaction.

A key notion in session types is that of *duality*, originating in linear logic: processes P and Q can be composed in parallel only when throughout the course of the computation each output of P's is matched by a suitable input of Q's, and vice versa. In this case we say that P and Q are *dual*. Session types ensure that only dual processes are composed in parallel. Hence typability guarantees the absence of communication errors such as mismatched communication and deadlocks. A process Q, dual to P above, would have the session type

Q : `?Int.!Bool.?Bool.end`

Notice that for each action in P's type, we have the dual action in Q's type, e.g. an output of type `!Int` can be received by an input of type `?Int`.

Session types can be viewed as finite state automata, with edges classifying message send/receive actions, and constraining causality between message exchange. For example, the behaviour of the process P above corresponds to the following FSA:



Unlike traditional automata, session types are built up compositionally from program syntax.

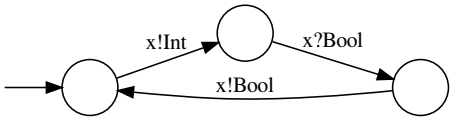
Recursive types, a key component of session types, allow for the description of protocols containing looping and repetition. Consider P', a variant of P above, that now instead of just sending an integer, receiving a boolean and sending a boolean, now performs those same three actions repeatedly. We denote such a session type:

P' : μX.!Int.?Bool.!Bool.X

or, alternatively:

P' : let X = !Int.?Bool.!Bool.X in X

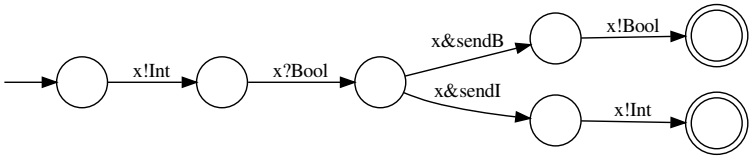
A corresponding finite state automaton for P' is shown below:



Further important components of session types are the dual notions of internal and external choice. Consider now our previous example P, further augmented thusly: instead of sending a boolean as the final action, the process now makes an *external choice*, where a selection is offered of either (1) sending a boolean as before, or (2) sending an integer. The dual process may select either option in an *internal choice*. We express this new behaviour with the session type P'':

P'' : !Int.?Bool.&⟨sendB:!Bool.end, sendI:!Int.end⟩

The tag sendB represents the boolean option, whereas sendI represents the integer option. This session type can be visualised by the following automaton:



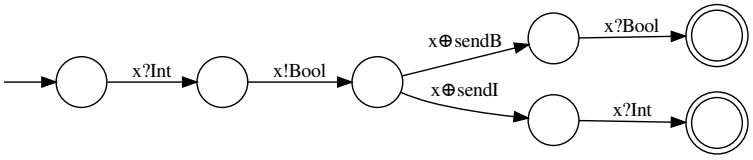
This external choice type represents a selection of services offered by a server P''. The dual process Q'' is therefore allowed to make an internal choice, whereby it selects from the services on offer. The process P'' must have a matching external choice of type S for every internal choice of type T that Q can make, such that S and T are dual. Note that the

converse does not always hold - while it is clear that we must never allow Q'' to select a service that P'' does not offer, we might allow P'' to offer a choice that Q'' never selects.

In the above case, we obtain the following dual type Q'':

Q'' : `?Int.!Bool.`$\oplus\langle$`sendB:?Bool.end, sentI:?Int.end`$\rangle$

This dual type corresponds to the following automaton:



LAST

Gay and Vasconcelos's calculus LAST(Lambda calculus with Asynchronous Session Types) [Gay and Vasconcelos, 2010] is a concurrent functional programming language, the first coherent integration of session types with lambda calculus. Processes (essentially functional programs) can be composed in parallel along with message buffers. Processes send messages that are placed in the message buffers, from where they are later asynchronously retrieved by other processes. Channels are held by processes and are used to specify which buffers receive which messages. Binary session types are imposed on the channels to ensure that communication patterns between these processes are always dual. Message exchange is achieved via `send` and `receive` combinators, which have the following types:

```
send :: T → !T.S → S
receive :: ?T.S → T ⊗ S
```

These types follow standard intuition about the behaviour of the respective constructs. The `send` combinator takes two arguments - a message of type `T`, and a channel which expects to perform communication of type `!T.S`. The `send` combinator delivers the message of type `T` along the channel, which we see from the channels type that it expects. The value returned is a channel of type `S` - the behaviour that we expect from the channel after the send operation. The `receive` combinator takes just one argument of type `?T.S` - the channel on which the process expects to receive a message - and returns a pair `T ⊗ S` - the received message and a channel on which communication can continue.

In LAST, session types are imposed via linearity constraints on channel names: each channel is used exactly once, and the interaction subsequently continues on a channel returned by the previous interaction. If linear channel usage was not enforced, processes

could violate their session types in a number of ways, for example, by attempting to receive a single message twice, which would cause the receiver to wait indefinitely, or by neglecting to send a message required by the session type.

The usage pattern of performing an interaction on a channel, and then rebinding the returned channel for the next interaction is forced upon us by linear typing of channels. This results in LASTprograms containing many `let` constructs, since every interaction requires the binding of a new channel. This creates syntactic noise in programs, which is arguably undesirable. In chapter 3, we show how implicit functions can provide a solution to this problem.

Shown below is an example LASTprogram, in which two communication partners initiate a session (via the `accept` and `request` operations). Each communication partner is a process, which consists of an expression enclosed in ⟨ angular brackets ⟩. These processes are separated by the *parallel composition* operator `|`, which indicates that the processes run concurrently, and can communicate if they each have one of the two dual endpoints of a communication channel. One process sends the integer to the other, and the other replies with the incremented received integer.

```
⟨ let    c = request x      in
  let    c = send 10      c in
  let m, c = receive      c in m    ⟩   |
⟨ let    d = accept x       in
  let n, d = receive      d in
  let    d = send (n + 1) d in unit ⟩
```

The session type for the first process is `?Int.!Int.end` and the type for the second is `!Int.?Int.end`. The first reduction is the session initiation, which creates two buffers (one for each direction) that the processes use for communication. The dual endpoints `c` and `d` to a shared communication channel are created, which can be used by each process to access these buffers. In general, a buffer $\texttt{a} \rightarrow (\texttt{b}, q)$ stores messages in a queue q sent on the channel `b`, that are to be received on the channel `a`. Messages are appended to the right of q when sent, and removed from the left when received. Session initiation also creates a restriction over the names `c` and `d` to prevent interference in the session by other processes. The session initiation yields:

```
(νcd)(
⟨ let    c = send 10      c in
  let m, c = receive      c in m    ⟩   |
⟨ let n, d = receive      d in
  let    d = send (n + 1) d in unit ⟩   |
c → (d, ϵ)   |   d → (c, ϵ))
```

Subsequently the second process performs the operation `send 10 c`, placing the value `10` in the buffer for the channel `d`:

```
(νcd)(
⟨ let m, c = receive      c in m    ⟩   |
⟨ let n, d = receive      d in
  let    d = send (n + 1) d in unit ⟩   |
c → (d, ϵ)   |   d → (c, 10))
```

In the next reduction step, the second process retrieves the value `10` from its buffer, and it is substituted for `n`:

```
(νcd)(
⟨ let m, c = receive       c in m    ⟩   |
⟨ let    d = send (10 + 1) d in unit ⟩   |
c → (d, ϵ)   |   d → (c, ϵ))
```

At this point, the subexpression `(10 + 1)` reduces to `11` (we omit this step for brevity). Following this, the first process deposits the result in the buffer for channel `c`:

```
(νcd)(
⟨ let m, c = receive c in m ⟩   |
⟨ unit                      ⟩   |
c → (d, 11)   |   d → (c, ϵ))
```

Finally the second process retrieves the value `11` from the buffer.

```
(νcd)(⟨ 11 ⟩  |  ⟨ unit ⟩  |  c → (d, ϵ)  |  d → (c, ϵ))
```

Linearity, session types and the Pi Calculus

[Giunti and Vasconcelos, 2013] introduces Pi calculus with Linear channel types *and* Session Types (PLST). PLST can type interesting processes not typable in other session-typed pi calculi, whilst retaining standard properties for session types such as deadlock-freedom and data race-freedom for linear resources. In particular, PLST allows for linear-to-unrestricted channel type evolution, meaning that a channel can have linearity constraints for a portion of a session, and evolve to become unrestricted later. This property is key for the following example, from the paper [Giunti and Vasconcelos, 2013].

Consider the following protocol P for a service provider for an online petition service: when a user wants to create an online petition, the petition service provider sends them a channel p, on which the creator sends the service provider the title, description and due date of the petition. The creator then sends their signature, after which they can distribute the channel p to potential signatories on the channels a_i. The process P describes the behaviour of the petition creator.

$$\begin{aligned} P = \; & onlinePetition(p).\overline{p}\langle title\rangle.\overline{p}\langle description\rangle.\overline{p}\langle dueDate\rangle \\ & .\overline{p}\langle signature\rangle.(\overline{a_1}\langle p\rangle \mid \ldots \mid \overline{a_n}\langle p\rangle) \end{aligned}$$

The process Q describes the protocol from the petition service's point-of-view. Q's input of signatures is replicated to allow for all signatories to deliver their signatures to the petition service. The process *ProcessSignatures* handles received signatures appropriately.

$$\begin{aligned} Q = \; & !(\nu p)\overline{onlinePetition}\langle p\rangle.p(title).p(description).p(duedate) \\ & .!p(signature).ProcessSignatures \end{aligned}$$

We obtain the following types T_1 and T_2 for p, T_1 representing Q's interaction with p, and T_2 representing the interaction of P and other signatories.

$$\begin{aligned} T_1 &= \texttt{lin?string.lin?string.lin?date.un?string.end} \\ T_2 &= \texttt{lin!string.lin!string.lin!date.un!string.end} \end{aligned}$$

Crucially, we see that the types begin with linear interaction, and after the transmission of date, accept arbitrary signatures.

The type system of PLST makes use of concepts seen in other calculi described in this literature review, such as a predicate $un(\delta)$ that is true when a type delta is unrestricted, context splitting rules, and standard session type duality with the property that $!T.S$ is dual to $?T.S'$ when S is dual to S'.

Multiparty Session Types

The session types we have seen thus far have been *binary* session types, in which there are exactly two participants whose actions are dual to one another. It is possible to generalise this binary form of session to an arbitrary number of participants, a so-called *multiparty* session. Consider the following protocol with three participants, with *participant numbers* `1`, `2` and `3`:

- First, participant `1` sends an integer to participant `2`.
- Then, participant `2` sends a boolean to participant `3`.
- Finally, participant `3` sends a string to participant `1`.

Instead of looking at this protocol from two dual points of view, we describe it with a type that takes a *global* view of all communication. We describe the above protocol with the following *global type* G:

`1→2:⟨Int⟩.2→3:⟨Bool⟩.3→1:⟨String⟩.end`

In the above, our communication types are of the form $\texttt{p} \to \texttt{q} : \langle T \rangle$, which tells us a message is sent by `p`, received by `q`, and that the message content is of type T.

It is possible to view this protocol from the point of view of any of the three participants, or in other words, to *project* the global type G onto a participant `p` ($1 \leq \texttt{p} \leq 3$), obtaining a *local* session type. We denote this projection $G \upharpoonright \texttt{p}$. In this case, $G \upharpoonright \texttt{1}$ yields:

$!\langle 2, \texttt{Int}\rangle.?\langle 3, \texttt{String}\rangle.\texttt{end}$

$G \upharpoonright 2$ yields:

$?\langle 1, \texttt{Int}\rangle.!\langle 3, \texttt{Bool}\rangle.\texttt{end}$

$G \upharpoonright 3$ yields:

$?\langle 2, \texttt{Bool}\rangle.!\langle 1, \texttt{String}\rangle.\texttt{end}$

In the local view we need only write a single participant number, since the participation of the local participant is implied. We write ! or ? to indicate whether the local participant does the sending or the receiving.

We can further project local session types onto another participant q, to obtain a type that describes only the communication between p and q. The syntax of such types matches the syntax of binary session types. Projecting a second participant number onto the above protocol leads to the following types:

$$G \upharpoonright 1 \upharpoonright 2 = !\texttt{Int.end} \qquad G \upharpoonright 1 \upharpoonright 3 = ?\texttt{String.end}$$
$$G \upharpoonright 2 \upharpoonright 1 = ?\texttt{Int.end} \qquad G \upharpoonright 3 \upharpoonright 1 = !\texttt{String.end}$$
$$G \upharpoonright 2 \upharpoonright 3 = !\texttt{Bool.end} \qquad G \upharpoonright 3 \upharpoonright 2 = ?\texttt{Bool.end}$$

Now that we have recovered binary session types by two projections, observe that for all participants p, q in G, $G \upharpoonright \texttt{p} \upharpoonright \texttt{q}$ is dual to $G \upharpoonright \texttt{q} \upharpoonright \texttt{p}$. This property holds for the above example, and is a condition of typability for all multiparty session-typed programs.

2.5 Scala and DOT

2.5.1 Dependent Object Types (DOT)

Dependent Object Types (DOT) is a programming language calculus first introduced in [Amin, Moors, and Odersky, 2012] in an unsound form. It was later refined, working towards a sound formation in [Amin, Rompf, and Odersky, 2014; Amin et al., 2016; Rompf and Amin, 2016]. DOT is a foundational calculus intended as a step towards a theoretical foundation for the programming language Scala and its type system.

Scala's main goal is scalability, and claims to achieve this by unifying the concepts of objects and module systems. DOT follows Scala in this regard, and can be thought of as an object calculus similar to [Abadi and Cardelli, 1994]. DOT models a restricted subset of Scala – the base-calculus includes Scala's key features from a type theory perspective, such as path-dependency, abstract type members and a subtyping hierarchy with maximal type $\top$ and minimal type $\bot$. Omitted are features of Scala such as traits, classes and inheritance. The table below (from [Rompf and Amin, 2016]) shows adaptations of modelled Scala constructs into DOT.

Scala	DOT
`{ type T = Elem }`	$T : \texttt{Elem..Elem}$
`{ type T>:S<:U }`	$T : S..U$
`{ def m(x:T) = t }`	$m(x) = t$
`A & B, A | B`	$A \wedge B, A \vee B$

Dot objects consist of lists of definitions d, with self-references bound by a variable. We write $\{z \Rightarrow \overline{d}\}$, where $\overline{d}$ has self-references z. Definitions d are type members $A = S..U$ or method members $m(x) = t$ where S, U are types, m and A are method and type labels respectively, x are variable names and t are terms. Terms t are then variables x, objects $\{z \Rightarrow \overline{d}\}$ or method invocations $t.m(t)$.

Abstract Type Members. Abstract type members are a feature of Scala and DOT that allow for generic programming. In Scala, a trait, class or object may declare an abstract type member. The trait, class or object may then declare or define methods over that type. Below is a trait `A` with an abstract type member `B`. The trait also contains a method member `consumeB` that consumes a value of type `B`.

```
trait A {
    type B
    def consumeB(b: B): String
}
```

Any object may then declare itself a subtype of `A`, and provide a concrete type in place of the abstract type member. Such an object may implement method members over that type:

```
object C extends A {
    type B = Int
    def consumeB(b: Int): String =
      b.toString
}
```

The above pattern is possible in DOT as well as Scala. We write $\{A : S..S'\}$ to denote an object with a type member A, whose lower bound is S and upper bound is S'. Scala's syntax `type T = U` is then equivalent to DOT's $\{T : U..U\}$ - we let `U` be the upper and lower bound. In DOT we can define a completely abstract type member (equivalent to just `type T` in Scala) by using $\bot$ and $\top$ as lower and upper bounds respectively. Conversely we can define a fully specified type member by using a single specified type as both the lower and upper bound. The above example is therefore translated into DOT as follows:

```
{ z ⇒
    A = { B: ⊥..⊤; consumeB(b) = b.toString() }
    C = A ∧ { B: Int..Int; consumeB(b) = b.toString() }
}
```

We enclose the declarations in an object with self-variable `z` as is required by DOT's syntax.

Path-dependent Types. Path-dependent types are in some sense the dual feature of abstract type members – they allow the referencing of an object's type members from other locations. Path-dependent types are also a restricted form of dependent types [Amin et al., 2016]. Instead of allowing arbitrary computations over values in types, objects with type members are the only values allowed, and selection of type members is the only permitted operation on those objects.

Path-dependent types with function arrows can be used to recover Hindley-Damas-Milner polymorphism despite the absence of type variables, via the passing of an object with a type member [Amin et al., 2016]. Consider the function `id` in a Haskell-like language:

```
id :: a → a
id x = x
```

We can rewrite this in DOT, with an additional parameter used to pass the type of the parameter x. The object $\{\texttt{A} : \bot..\top\}$ stands in for a type variable. The object's member A is a label given to an unspecified type, i.e. a type whose lower and upper bounds are $\bot$ and $\top$ respectively. The encoding of a polymorphic `id` in DOT is given below:

```
{ z ⇒
    id(u: {A: ⊥..⊤})(x: u.A): u.A
    id(u)(x) = x
}
```

As a further example we show the sorting function from section 2.3.3 adapted to DOT. Again path-dependency is used to stand in for the type variables of Hindley-Damas-Milner polymorphism. The type parameter `x` is passed to the `isort` function and is not used (except in being passed to recursive calls) as its sole purpose is to make the program type-check. The parameter `x` in insert, however, functions as both a type and value parameter. The example is given below:

```
{ z ⇒
    insert(x: {A: ⊥..⊤})(l: List ∧ {T = x.A})
        (f: Comparator ∧ {T = x.A}): List ∧ {T = x.A}
    insert(x)(l)(f) = if (l.isEmpty) then x :: l
        else if f(x)(l.hd) then x :: l
        else l.hd :: z.insert(x)(l.tl)(f)
    isort(x: {A: ⊥..⊤})(l: List ∧ {T = x.A})
        (f: Comparator ∧ {T = x.A}): List
    isort(x)(l)(f) = if (l.isEmpty) then List.empty
        else z.insert(l.hd)(z.isort(x)(l.tl)(f))(f)
}
```

2.6 Summary

In this literature review, we have studied the topic of type systems, and seen their applications in lambda, pi and other calcului. We have explored how logical inference rules (typing rules) are used to determine which programs are well-behaved and which are not, based on semantics. We have seen various models of concurrency, and how seen how types can be used not just to enforce that values are passed sensibly, but to enforce that the patterns of passing are themselves sensible. Finally we introduced the calculus DOT, which serves as a theoretical foundation for the popular programming language Scala.

In the next chapter, we build on these foundations by introducing a novel programming language feature inspired by implicit functions and based on session types: *implicit messages*.

Chapter 3

Asynchronous Sessions with Implicit Functions and Messages[1]

3.1 Introduction

This chapter introduces the calculus IM, a concurrent lambda calculus based on LAST, integrating implicit functions, and a novel implicit program construct: implicit messages. Implicit messages are a generalisation of implicit functions to concurrent computation. Implicit functions can be summarised: a function parameter is marked as implicit in the function's type, the implicit parameter is then used in its body, and callers may omit the parameter, the compiler filling it in later. Implicit messages are analogously summarised: a message in a session is marked as implicit in the session type, the implicit message is then used in the receiver's body, and senders may omit the parameter, the compiler filling it in later. Implicit messages allow us to translate use cases for implicit functions into concurrent analogues, such as dependency injection and ad-hoc polymorphism. Implicit functions allow dependency injection by inserting implicit context variables into function calls that require context; implicit messages allow dependency injection by inserting implicit context variables into client-to-server messages where the server requires context. Implicit functions achieve ad-hoc polymorphism (type classes) by passing a dictionary of functions that implement behaviours for a specific type; implicit messages achieve ad-hoc polymorphism (*session* type classes) by passing a reference to services that implement behaviours for a specific type.

IM's semantics are derived from a translation to LAST, in which implicit functions and messages are made explicit. We prove IM type-safe via this translation, showing that it preserves types.

[1]This chapter is adapted from [Jeffery and Berger, 2018] and [Jeffery and Berger, 2019], published works co-authored with my supervisor, Dr. Martin BERGER. All proofs and figures are my own work. I estimate that 90% of the prose in this chapter is completely my own work, with the last 10% being co-written.

3.1.1 Outline

Section 3.2 of this chapter introduces IM by example, showing use cases for implicit functions and messages in a LAST-like language. Section 3.3 introduces IM's term syntax, section 3.4 introduces IM's types, section 3.5 introduces the translation from IM to LAST, and section 3.6 deals with type safety.

3.2 IM - Examples

3.2.1 Elimination of repeated rebinding

A well-known problem with the integration of session types and sequential languages is the seeming necessity of repeated rebinding of channel names. The problem is that `send` takes a channel of type $!T.S$ as its second argument, and returns a linear channel of type S. In order for linearity to be respected that channel must be rebound. Consider the process below, typical of LAST programs. Note that the `select` combinator corresponds to the type $\oplus\langle \ldots \texttt{l:S}, \ldots \rangle$ introduced in section 2.4.8, and the process below selects from the paths `label1` or `label2` offered by its dual, based on the result of `pred(m)`.

```
miscService :: ⟨S⟩^a → end
miscService ap =
    let    c = accept ap in
    let m, c = receive c in
    let n, c = receive c in
    if pred(m) then
        let c = select label1 c in
        let c = send f(m, n) c in
        let c = send g(m, n, n) c in c
    else
        let c = select label2 c in
        let o, c = receive c in
        let c = send f(n, m) c in
        let c = send g(m, n, o) c in c
```

This redundancy makes programs hard to read. The issue can be addressed in other ways, for example using parameterised monads [Atkey, 2009], see also [Gay and Vasconcelos, 2010, Chapter 7]. Implicit functions and message passing enable a principled and canonical two-step solution: (1) make the channel argument implicit and let the compiler synthesise the missing channel name for rebinding; (2) include in the language two special constructs: `let`$^{\wr}$, which unpacks a pair of form *(value, channel)*, such that the left hand value is bound to a given name, and the right hand channel is bound to the implicit variable; and `;`$^{\wr}$, which binds a single channel variable, resultant on the left, to the implicit scope of the computation on the right. This solution is sufficient for languages that are not known to admit monads, and requires only implicit functions.

The `send` primitive has type $T \to !T.S \to S$. We can use implicit function types to define a new output primitive $\texttt{send}^{\wr}$, with type $T \to !T.S \wr\to S$, explained below. The annotation $\wr$ in $!T.S \wr\to S$ makes the channel argument implicit - the message will be sent on a channel in the implicit scope with the appropriate session type.

```
send≀ :: T → !T.S ≀↦ S
send≀ m = send m ≀
```

We can do something similar for `select` and `receive`.

```
select≀ :: Label l → ⊕⟨...l:S,...⟩ ≀↦ S
select≀ l = select l ≀

receive≀ :: ?T.S ≀↦ T ⊗ S
receive≀ = receive ≀
```

We define $\texttt{let}^{\wr}$ and $;^{\wr}$ as follows:

$$\begin{aligned} \texttt{let}^{\wr}\, x = e_1 \,\texttt{in}\, e_2 &\stackrel{\text{def}}{=} \texttt{let}\, x, \wr = e_1 \,\texttt{in}\, e_2 \\ e_1 \,;^{\wr}\, e_2 &\stackrel{\text{def}}{=} \texttt{let}\, \wr = e_1 \,\texttt{in}\, e_2 \end{aligned}$$

We can rewrite `miscService` above with our new primitives. The resulting code is less repetitive and more terse, hence readable.

```
miscService :: ⟨S⟩ᵃ ≀↦ end
miscService =
    let ≀ = accept ≀ in
    let m = receive≀ in
    let n = receive≀ in
    if pred(m) then
        select≀ label1 ;≀
        send≀ f(m, n) ;≀
        send≀ g(m, n, n)
    else
        select≀ label2 ;≀
        let o = receive≀ in
        send≀ f(n, m) ;≀
        send≀ g(m, n, o)
```

We believe that it is possible to omit the superscript $\wr$ annotations, using type inference to distinguish ordinary `let` constructs from our new $\texttt{let}^{\wr}$ constructs. $;^{\wr}$ constructs could likely be omitted and programs appropriately augmented in a similar way. An implementation could conceivably try to type check the program with normal `let` and `;` constructs, and replace them with implicit ones if they cause the type check to fail. Scala's implicit conversions are implemented with such a heuristic [Sobral and Braun, 2011], and such a technique is likely applicable here.

Our solution to the rebinding problem is robust, and applicable to linear types generally. [Bernardy et al., 2017] presents a linear typing system for Haskell, and demonstrates the use of linear functions in Haskell. A prototypical use case of linear functions in Haskell is to prevent common file errors, such as writing to a closed file or double closure. Linear Haskell provides a set of functions that take as a linear parameter a file, which is then returned and rebound. Some examples of such functions and their types are given next:

```
openFile :: FilePath → IO_L 1 File
readLine :: File ⊸ IO_L 1 (File, Unrestricted Bytestring)
closeFile :: File ⊸ IO_L ω ()
```

Here IO_L is a type constructor that represents types obtained by doing IO, and the *multiplicities* 1 or ω denote whether or not the type is linear - 1 for linear, ω for unrestricted. A typical usage example of these functions might be:

```
do f <- openFile "myFile.txt"
   line, f <- readLine f
   if somePredicate line then
       line2, f <- readLine f
       closeFile f
       return (line ++ line2)
   else
       return line
```

Rewriting the functions `openFile`, `readLine` and `closeFile` in a similar manner to above yields the following types:

```
openFile^ϱ :: FilePath ϱ→ IO_L 1 File
openFile^ϱ = openFile ϱ

readLine^ϱ :: File ϱ⊸ IO_L 1 (File, Unrestricted Bytestring)
readLine^ϱ = readLine ϱ

closeFile^ϱ :: File ϱ⊸ IO_L ω ()
closeFile^ϱ = closeFile ϱ
```

We reuse our definition of $;^{\varrho}$, and define a special assignment operator $<-^{\varrho}$ similarly to `let`$^{\varrho}$, as follows:

$$x <-^{\varrho} e \stackrel{\text{def}}{=} \varrho, x <- e$$

With these components we can rewrite the above to the following:

```
do openFile⟨ "myFile.txt" ;⟨
   line <-⟨ readLine⟨
   if somePredicate line then
       line2 <-⟨ readLine⟨
       closeFile⟨ ;⟨
       return (line ++ line2)
   else
       return line
```

3.2.2 Session type classes

Type classes [Kaes, 1988; Wadler and Blott, 1989] provide type-safe ad-hoc polymorphism by means of constraints on parametrically polymorphic types. They allow the programmer to define a fixed set of functions over multiple datatypes, where each datatype has a bespoke implementation of each function in the set. We call these sets of functions type classes. They are usually implemented by *dictionary passing* [Wadler and Blott, 1989]. That means that at compile time an additional argument (the dictionary) and suitable access to this argument are synthesised for all code depending on type classes. With implicit arguments we can make dictionary passing implicit, and type classes become a special case of implicit arguments. This is a common Scala idiom [Oliveira, Moors, and Odersky, 2010].

Implicit messages suggest a natural generalisation of type classes: pass access to dictionaries by implicit messages! We illustrate this idea with a simple example. In Haskell, `Show` is a type class that converts values to their string representation. We generalise this to IM: instead of a conversion function, IM has a conversion *server*. We show two example implementations `intShow` and `boolShow` (we omit the details of the former). Additional function servers can be written against this code over types that define a `Show` type class server. The type `Show` itself should be interpreted as a type schema, much like the types of the `send` and `receive` combinators.

```
type Show = ?a.?ˡ⟨?a.!String.end⟩ᵃ.!String.end

show :: ⟨Show⟩ᵃ → end
show c =
    let         c = accept c           in
    let a     , c = receive c          in
    let aShow , c = implicit receive c in
    let         d = request aShow      in
    let         d = send a d           in
    let as    , d = receive d          in
    send as c

implicit boolShow :: ⟨?Bool.!String.end⟩ᵃ → end
boolShow c =
    let     c = accept c  in
    let b , c = receive c in
    send (if b then "true" else "false") c

implicit intShow :: ⟨?Int.!String.end⟩ᵃ → end
intShow = ...

showUser :: ⟨Show⟩ʳ → end
showUser ap =
    let    c = request ap in
    let    c = send 10 c  in
    let s, c = receive c  in
    printf(s) ;
    c
```

Clients communicating with the `show` server such as `showUser` do not need explicitly to send their `show` implementation, but send one implicitly.

It would be possible to make this example even more terse by eliminating repeated rebinding with implicit functions, however for clarity we exhibit just one application of implicits at a time.

3.2.3 Context and dependency injection

Implicit functions are commonly used in Scala to pass contextual information to a large set of methods. If many methods require the same contextual information, explicitly passing this context to each method call as a parameter becomes laborious, and eliding this contextual information becomes desirable. It is a common Scala idiom to use implicit functions to elide this repetitious context passing.

This pattern can now easily be generalised to concurrency by eliding contextual information passing in client/server interaction. The following example is an implementation of a simple web server, that receives a request from a client and dispatches each kind of

request to an appropriate handler. Some contextual information is passed to each handler at the handler's dispatch time - this information might typically be a network configuration or database reference. An implementation without implicit messages requires the context to be passed to each handler each time one is spawned. This creates syntactic noise that the programmer might prefer to elide. It also introduces repetition which can lead to programmer error. Implicit messages allow us to omit this context passing by passing the context as an implicit message, decreasing repetition and thereby reducing the cognitive burden on the programmer.

Each handler eventually sends a response to the manager, with the result of its computation. The result can also be passed implicitly, further reducing the syntactic noise.

We show an implementation without implicit messages on the left, and an implementation with implicit messages on the right.

```
type Manager = &{
    service1: H̄1
    ...
    serviceN: H̄n
}
manager :: ⟨Manager⟩ → Ctx → end
manager c ctx = case c of {
    service1:
        let d = request handler1 in
        let d = send ctx d in
        ...
        let res, d = receive d in
        manager ctx
    ...
    serviceN:
        let d = request handlerN in
        let d = send ctx d in
        ...
        let res, d = receive d in
        manager ctx
}
handler1Impl :: H1
handler1Impl =
    let d = accept handler1 in
    let ctx, d = receive d in
    ...
    let res = ... in
    let d = send res d in
...
handlerNImpl :: Hn
handlerNImpl =
    let d = accept handlerN in
    let ctx, d = receive d in
    ...
    let res = ... in
    let d = send res d in
```

```
type Manager = &{
    service1: H̄1
    ...
    serviceN: H̄n
}
manager :: ⟨Manager⟩ → Ctx ↬ end
manager c = case c of {
    service1:
        let d = request handler1 in
        ...
        let res, d = implicit receive d in
        manager

    ...
    serviceN:
        let d = request handlerN in
        ...
        let res, d = implicit receive d in
        manager

}
handler1Impl :: H1
handler1Impl =
    let d = accept handler1 in
    let ↿, d = implicit receive d in
    ...
    let ↿ = ... in d

...
handlerNImpl :: Hn
handlerNImpl =
    let d = accept handlerN in
    let ↿, d = implicit receive d in
    ...
    let ↿ = ... in d
```

3.3 The language IM

This section presents the syntax of our language IM of implicit message passing. IM is a superset of LAST. LAST is a medium through which the idea of implicit message passing can be expressed. Its integration of functions and processes enables us to provide both: implicit functions and implicit messages.

As the compiler synthesises the missing arguments at compile-time from type information, calculi for implicit arguments might be best understood not as programming languages, but as meta-programming systems that generate code in a base language L from input programs in L with implicits. Indeed, SI [Odersky et al., 2018], an extension of System F, Scala's foundations for implicits, does not have a self-contained operational semantics, and is instead compiled to System F. We use the same approach, and translate IM to LAST.

3.3.1 Syntax

In the presentation of IM's syntax, let v range over values and e over expressions. We assume that x ranges over a countable set of term variables, c over a countable set of channel endpoints, n over $\mathbb{N} \cup \{\infty\}$, l over labels and I over finite subsets of $\mathbb{N}$. In order to make the presentation easily accessible, we highlight the extensions IM adds to LAST. The grammar of IM expressions is given in figure 3.1.

$$
\begin{array}{rcl}
v & ::= & \lambda x.e \mid (v, v) \mid \texttt{unit} \mid \texttt{fix} \mid \texttt{fork} \\
& \mid & \texttt{request}\ n \mid \texttt{accept}\ n \mid \texttt{send} \\
& \mid & \texttt{receive} \mid \texttt{implicit receive} \\
e & ::= & v \mid e\ e \mid (e, e) \mid \texttt{let}\ x, x = e\ \texttt{in}\ e \\
& \mid & \texttt{select}\ l\ e \mid \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} \\
& \mid & \wr \mid \texttt{let}\ x, \wr = e\ \texttt{in}\ e
\end{array}
$$

FIGURE 3.1: Grammar of IM expressions

Here `implicit receive` is the implicit analog of `receive`. Unlike `receive`, it is not matched by a corresponding `send`, but a corresponding `send` is inserted during translation, while `implicit receive` is translated into a normal `receive`. $\wr$ denotes a query to the implicit scope. $\wr$ is removed at translation time, and is replaced by a nondeterministically chosen name in the implicit scope. This nondeterminism can be resolved via relatively simple heuristics, some of which are discussed in section 3.5.1. The construct `let x, ≀ = ...` allows us to add variables to the implicit scope, and as with the lone $\wr$, we also replace $\wr$ within `let` by a variable name during translation. Note that we often write `let ≀ = e in e'`. This is a convenience and can be thought of as syntactic sugar for `let _, ≀ = (_, e) in e'` where _ is an unused variable or expression.

The parameter n following `accept` n and `request` n gives a *bound* for session communication. This will be explained in later sections. Note that we omit the bound parameter for brevity where not relevant.

An IM program is a configuration of expressions in parallel, running as separate threads and typed in a suitable environment. We define the syntax of configurations, ranged over by C, in figure 3.2.

$$
\begin{array}{rcl}
b & ::= & v \mid l \\
C & ::= & C \parallel C \mid c \mapsto (c, n, \vec{b}) \mid (\nu cc)C \mid \langle e \rangle
\end{array}
$$

FIGURE 3.2: Grammar of IM configurations

3.3.2 Semantics

We derive semantics for IM by translating IM programs into LAST, and leverage LAST's semantics, which are given in figures 3.3 (expressions) and 3.4 (configurations). Figure 3.5 defines structural congruence for LAST, on which depend the semantics. Contexts E, E', ... are LAST expressions with a single hole, and $E[e]$ denotes filling the hole in the context E with the expression e.

$$(\lambda x.e)\ v \longrightarrow_v e[v/x] \quad \text{[R-App]} \qquad \texttt{fix}\ (\lambda x.e) \longrightarrow_v e[(\texttt{fix}\ (\lambda x.e))/x] \quad \text{[R-Fix]}$$

$$\texttt{let}\ x, y = (v, u)\ \texttt{in}\ e \longrightarrow_v e[v/x][u/y] \quad \text{[R-Split]}$$

FIGURE 3.3: Semantics of LAST expressions

$$\frac{e \longrightarrow_v e'}{\langle E[e] \rangle \longrightarrow \langle E[e'] \rangle} \quad \text{[R-Thread]} \qquad \frac{C \longrightarrow C'}{C \parallel C'' \longrightarrow C' \parallel C''} \quad \text{[R-Par]}$$

$$\langle E[\texttt{fork}\ e] \rangle \longrightarrow \langle e \rangle \parallel \langle E[\texttt{unit}] \rangle \quad \text{[R-Fork]}$$

$$\frac{C \longrightarrow C'}{(\nu cd)C \longrightarrow (\nu cd)C'} \quad \text{[R-New]}$$

$$\frac{C \equiv C' \quad C' \longrightarrow C'' \quad C'' \equiv C'''}{C \longrightarrow C'''} \quad \text{[R-Struct]}$$

$$\langle E[\texttt{request}\ \ n\ x] \rangle \parallel \langle E'[\texttt{accept}\ \ n'\ x] \rangle \longrightarrow (\nu cd)(c \mapsto (d, n, \epsilon) \parallel d \mapsto (c, n', \epsilon) \parallel \langle E[c] \rangle \parallel \langle E'[d] \rangle) \quad \text{[R-Init]}$$

$$c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}) \parallel \langle E[\texttt{send}\ v\ c] \rangle \longrightarrow c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}v) \parallel \langle E[c] \rangle \text{ if } |\vec{b}| < n \quad \text{[R-Send]}$$

$$c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}) \parallel \langle E[\texttt{select}\ l\ c] \rangle \longrightarrow c \mapsto (d, n', \vec{b'}) \parallel d \mapsto (c, n, \vec{b}l) \parallel \langle E[c] \rangle \text{ if } |\vec{b}| < n \quad \text{[R-Select]}$$

$$c \mapsto (d, n, v\vec{b}) \parallel \langle E[\texttt{receive}\ c] \rangle \longrightarrow c \mapsto (d, n, \vec{b}) \parallel \langle E[(v, c)] \rangle \quad \text{[R-Receive]}$$

$$c \mapsto (d, n, l_j\vec{b'}) \parallel \langle E[\texttt{case}\ c\ \texttt{of}\ \{l_i : e_i\}_{i \in I}] \rangle \longrightarrow c \mapsto (d, n, \vec{b}) \parallel \langle E[e_j\ c] \rangle \text{ if } j \in I \quad \text{[R-Branch]}$$

FIGURE 3.4: Semantics of LAST configurations

$$C_1 \parallel C_2 \equiv C_2 \parallel C_1 \quad \text{[E-Comm]} \qquad C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3 \quad \text{[E-Assoc]}$$
$$C_1 \parallel (\nu cd)C_2 \equiv (\nu cd)C_1 \parallel C_2 \text{ if } c, d \notin fc(C_1) \quad \text{[E-Scope]}$$

FIGURE 3.5: Structural congruence for LAST

3.4 Types for IM

Just as SI is given meaning by type-guided translation to System F in [Odersky et al., 2018], we give such a translation of IM into LAST. This section prepares the translation by extending LAST's typing system with types for implicit message passing and implicit functions. Types for IM are given by the following grammar. Here T ranges over types for the lambda calculus part of IM, S over session types, and B over buffer types.

$$\begin{array}{rcl} T & ::= & \texttt{Unit} \mid S \mid T \otimes T \mid T \to T \mid T \multimap T \\ & \mid & \langle S \rangle^{\mathrm{r}} \mid \langle S \rangle^{\mathrm{a}} \mid \langle S, S' \rangle \mid T \wr\!\!\to T \\ & \mid & T \wr\!\!\multimap T \\ S & ::= & \texttt{end} \mid ?T.S \mid !T.S \mid \&\langle l_i : S_i \rangle_{i \in I} \\ & \mid & \oplus\langle l_i : S_i \rangle_{i \in I} \mid X \mid \mu X.S \mid ?^{\wr}T.S \\ & \mid & !^{\wr}T.S \\ B & ::= & T \mid l \end{array}$$

FIGURE 3.6: Grammar of IM types

Note that we consider only tail-recursive session types, rejecting those whose form is, for example, $\mu S.!S.S$, where a recursion variable is used as message contents, as reasonable definitions of duality for such session types are unclear [Lindley and Morris, 2016].

The type $T \wr\!\!\to T$ is the type of implicit functions. It is written $?\to$ in [Odersky et al., 2018] but we replace $?$ by $\wr$ to avoid confusion with the input session type $?T.S$. The type $T \wr\!\!\multimap T$ is the linear equivalent of $T \wr\!\!\to T$. As with [Odersky et al., 2018], we do not have syntax for implicit abstraction and application - these are inferred during *implicit resolution* in Section 3.5.

The types $!^{\wr}T.S$ and $?^{\wr}T.S$ are the types of implicit message input and output respectively. They are the dual of one another as with explicit output and input. Implicit output types cannot be deduced from a process's syntax (since they are implicit) and must be inferred by inspecting the process that contains the corresponding implicit input. This happens during implicit resolution.

Buffer content types $\vec{B}$ are composed of vectors of entries B. Each entry is either a type T, representing the type of a value that is to be sent and stored in the buffer, or a label l representing the selection of such an option l by a process communicating using

the buffer. Buffer content types $\vec{B}$ are assigned to buffers $\vec{b}$ such that for each v in $\vec{b}$ there exists a type T in the corresponding buffer content type $\vec{B}$ such that $v : T$. This notion is made precise in Section 3.5.

Type schemas for constants

Given in figure 3.7 are the type schemas for the constants k. They are the same as LAST's, and can be instantiated for any appropriate type.

$$
\begin{array}{lcl}
\texttt{fix} & : & (T \to T) \to T \\
\texttt{send} & : & T \to !T.S \multimap S \\
\texttt{send} & : & T \to !T.S \to S \quad \text{if } un(T) \\
\texttt{fork} & : & T \to \texttt{Unit} \quad \text{if } un(T) \\
\texttt{receive} & : & ?T.S \to T \otimes S \\
\texttt{request}\ n & : & \langle S \rangle^{\mathrm{r}} \to \overline{S} \quad \text{if } bound(\overline{S}) \leq n \\
\texttt{accept}\ n & : & \langle S \rangle^{\mathrm{a}} \to S \quad \text{if } bound(S) \leq n \\
\texttt{unit} & : & \texttt{Unit}
\end{array}
$$

FIGURE 3.7: Type schemas for IM constants

Note that we omit a type schema for `implicit receive`. This is because it cannot be translated by the rule [T-CONST] in Figure 3.14, but needs a bespoke typing rule as unlike the other constants its translation is not identity.

Session type duality

We give the session type duality function for our calculus in figure 3.8. If a session type S and S' are *dual*, written $\overline{S} = S'$, then a pair of terms of types S and S' can interact without communication errors. Such processes match in the sense that every action that one takes is matched by the other. If one outputs, the other inputs. If one offers a choice, the other makes a choice. We extend duality function of LAST to include the two forms of implicit communication.

$$
\begin{array}{rclcrcl}
\overline{?T.S} & = & !T.\overline{S} & \qquad & \overline{!T.S} & = & ?T.\overline{S} \\
\overline{?^{\iota}T.S} & = & !^{\iota}T.\overline{S} & & \overline{!^{\iota}T.S} & = & ?^{\iota}T.\overline{S} \\
\overline{\mu X.S} & = & \mu X.\overline{S} & & \overline{X} & = & X \\
\overline{\oplus\langle l_i : S_i\rangle_{i\in I}} & = & \&\langle l_i : \overline{S}_i\rangle_{i\in I} & & \overline{\texttt{end}} & = & \texttt{end} \\
\overline{\&\langle l_i : S_i\rangle_{i\in I}} & = & \oplus\langle l_i : \overline{S}_i\rangle_{i\in I} & &
\end{array}
$$

FIGURE 3.8: Duality for IM session types

We define the subtyping for IM coinductively by extension of the definition for LAST.

Definition 1. A type T is *contractive* if it does not have subexpressions of the form $\mu X_1 ... \mu X_n . X_i$ where $0 < i \leq n$.

Let $\mathscr{S}$ denote the set of contractive, closed session types, and let $\mathscr{T}$ denote the set of types in which all session types are contractive and closed. We define the function $F(\cdot)$ on binary relations over $\mathscr{T}$ in figure 3.9.

$$
\begin{aligned}
F(R) = {} & \{(\mathtt{end}, \mathtt{end})\} \\
& \cup \{(?T.S, ?T'.S') | (T, T'), (S, S') \in R\} \\
& \cup \{(!T.S, !T'.S') | (T', T), (S, S') \in R\} \\
& \cup \{(?^{\wr} T.S, ?^{\wr} T'.S') | (T, T'), (S, S') \in R\} \\
& \cup \{(!^{\wr} T.S, !^{\wr} T'.S') | (T', T), (S, S') \in R\} \\
& \cup \{(\&\langle l_i : S_i \rangle_{i \in I}, \&\langle l_j : S'_j \rangle_{j \in J}) \\
& \qquad | I \subseteq J, (S_i, S'_i) \in R, \forall i \in I\} \\
& \cup \{(\oplus\langle l_i : S_i \rangle_{i \in I}, \oplus\langle l_j : S'_j \rangle_{j \in J}) \\
& \qquad | J \subseteq I, (S_i, S'_i) \in R, \forall i \in J\} \\
& \cup \{(\langle S, S' \rangle, \langle S \rangle^{\mathrm{a}}) | S, S' \in \mathscr{S}\} \\
& \cup \{(\langle S, S' \rangle, \langle S' \rangle^{\mathrm{r}}) | S, S' \in \mathscr{S}\} \\
& \cup \{(\langle S \rangle^{\mathrm{a}}, \langle S' \rangle^{\mathrm{a}}) | (S, S') \in R\} \\
& \cup \{(\langle S \rangle^{\mathrm{r}}, \langle S' \rangle^{\mathrm{r}}) | (S, S') \in R\} \\
& \cup \{(\langle S_1, S'_1 \rangle, \langle S_2, S'_2 \rangle) | (S_1, S_2), (S'_1, S'_2) \in R\} \\
& \cup \{(T \to T', T \multimap T') | T, T' \in \mathscr{T}\} \\
& \cup \{(T_1 \to T'_1, T_2 \to T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
& \cup \{(T_1 \multimap T'_1, T_2 \multimap T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
& \cup \{(\mu X.S, S') | (S[\mu X.S / X], S') \in R\} \\
& \cup \{(S, \mu X.S') | (S, S'[\mu X.S' / X]) \in R\} \\
& \cup \{(T_1 \wr\!\!\to T'_1, T_2 \wr\!\!\to T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
& \cup \{(T_1 \wr\!\!\multimap T'_1, T_2 \wr\!\!\multimap T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
& \cup \{(T \wr\!\!\to T', T \wr\!\!\multimap T') | T, T' \in \mathscr{T}\} \\
& \cup \{(T \to T', T \wr\!\!\to T') | T, T' \in \mathscr{T}\} \\
& \cup \{(T \multimap T', T \wr\!\!\multimap T') | T, T' \in \mathscr{T}\} \\
& \cup \{(!T.S, !^{\wr} T.S) | T \in \mathscr{T}, S \in \mathscr{S}\}
\end{aligned}
$$

Figure 3.9: Subtyping for IM types

Contractivity ensures that F is monotone. We write $T <: U$ if the pair (T, U) is in the greatest fixpoint of F. The last three lines in the definition of $F(\cdot)$ allow us to type the use of an explicit function in place of an implicit one, and the sending of explicit messages

to implicit inputs. Such behaviour is allowed in Scala - the user may pass an explicit argument to an implicit function. We allow the same behaviour with implicit functions, and the analogous behaviour in the case of implicit messages.

The *matches* relation determines whether a given buffer type $\vec{B}$ agrees with a session type S. We write $\vec{B}\ mat\ S$ when the types in $\vec{B}$ match a prefix of those in S. We formalise this notion with the rules in figure 3.10.

$$\frac{\vec{B}\ mat\ S \quad U <: T}{U\vec{B}\ mat\ ?^{\wr}T.S}\ [\text{M-OUTI}] \qquad \frac{\vec{B}\ mat\ S \quad U <: T}{U\vec{B}\ mat ?T.S}\ [\text{M-OUT}]$$

$$\frac{}{\epsilon\ mat\ S}\ [\text{M-EMPTY}] \qquad \frac{\vec{B}\ mat\ S}{l\vec{B}\ mat\ \&\langle ..., l : S, ...\rangle}\ [\text{M-CASE}]$$

FIGURE 3.10: The *mat* relation for IM

For some S and $\vec{B}$ such that $\vec{B}\ mat\ S$, $S/\vec{B}$ gives the session behaviours remaining as a *postfix* of S after performing those behaviours that correspond with $\vec{B}$. We define the postfix operator in figure 3.11.

$$S/\epsilon = S \qquad ?T.S/U\vec{B} = S/\vec{B}$$
$$?^{\wr}T.S/U\vec{B} = S/\vec{B} \qquad \&\langle ..., l : S, ...\rangle/l\vec{B} = S/\vec{B}$$

FIGURE 3.11: The postfix operator for IM

Session type bounds

We define $bound(S)$, which gives the *bound* of a session type, an upper bound on the runtime size of the buffer required to hold the values received on a channel with session type S. We start with the auxiliary operator $\mathsf{bds} \in (\mathscr{S} \to \mathbb{N}^{\infty}) \to \mathscr{S} \to \mathbb{N}^{\infty}$, defined in 3.12.

$$\mathsf{bds}(f)(S) = \begin{cases} 1 + f(S') & S \in \{?T.S', ?^{\wr}T.S'\} \\ 1 + max\{f(S_i)\}_{i \in I} & S = \&\langle l_i : S_i\rangle_{i \in I} \\ f(S[^{\mu X.S'}/_X]) & S = \mu X.S' \\ 0 & \text{otherwise} \end{cases}$$

FIGURE 3.12: The bds operator for IM

We define the relation $S \mapsto S'$, which computes an *advanced* session type S' given a session type S in figure 3.13.

Finally we define $bound(S) = max\{\mu(S') | S \mapsto^* S'\}$ where μ is the least fixed point of bds.

$$?T.S \mapsto S \qquad !T.S \mapsto S$$
$$?^{\wr}T.S \mapsto S \qquad !^{\wr}T.S \mapsto S$$
$$\&\langle ..., l : S, ... \rangle \mapsto S \qquad \oplus \langle ..., l : S, ... \rangle \mapsto S$$
$$\mu X.S \mapsto S' \text{ if } S\{\mu X.S/X\} \mapsto S'$$

FIGURE 3.13: The $\mapsto$ operator for IM

3.5 Translation from IM to LAST

This section presents implicit resolution, the type-directed translation of IM programs to LAST. We proceed in three steps, translation of expressions, translation of buffers and translation of configurations. Following [Odersky et al., 2018], the translation is type-directed in that we give typing rules for IM, instrumented with translations to LAST. By forgetting the instrumentation, we obtain a typing system for IM.

Typing environments and implicit scope

Implicit resolution removes queries $\wr$ and inserts explicit functions and messages in place of implicit ones. This happens by choosing arguments from the implicit scope. We define the implicit scope thusly: The typing environment Γ is divided into two parts: the implicit and explicit scopes. That is to say, some of the bindings in Γ refer to implicit variables and some to explicit variables. In our typing rules we range over implicit variables with y and explicit variables with x. Variables enter the implicit scope in several ways: (1) when received as an implicit message; (2) when given as an argument to an implicit function; and (3) when bound by a `let` construct with $\wr$ on the left-hand side of the $=$.

Typing and translation of expressions

Typing judgements for expressions are of the form $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$. This can be read as: "under assumptions Γ, the IM expression e has type T and is translated to the LAST expression $\widehat{e}$". Our typing and translation rules can be found in Figure 3.14. With the exception of the new syntactic forms of expressions, the translations are homomorphic, yielding rules similar in structure to those found in [Gay and Vasconcelos, 2010]. The rules for our new syntactic forms are more interesting. The rules [T-SPLITI], [T-APPI], [T-ABSI] and [T-QUERY] follow a similar structure to those in [Odersky et al., 2018]. Note that with [T-QUERY], the variable chosen to replace $\wr$ must satisfy linearity constraints, a restriction not present in [Odersky et al., 2018]. [T-ABSLI] is a linear version of the rule for implicit functions and is effectively a combination of the rules [T-ABSI] and [T-ABSL]. The rule [T-INI] translates `implicit receive` into `receive` and otherwise behaves in

$$\frac{\Gamma \vdash e : T \rightsquigarrow \widehat{e} \quad T <: U}{\Gamma \vdash e : U \rightsquigarrow \widehat{e}}\ [\text{T-Sub}] \qquad \frac{un(\Gamma) \quad k : T}{\Gamma \vdash k : T \rightsquigarrow k}\ [\text{T-Const}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2, x_1 : T, x_2 : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x_1, x_2 = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x_1, x_2 = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2}\ [\text{T-Split}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \rightsquigarrow \widehat{e}_1 \quad \Gamma_2 \vdash e_2 : U \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : T \otimes U \rightsquigarrow (\widehat{e}_1, \widehat{e}_2)}\ [\text{T-Pair}]$$

$$\frac{\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e} \quad un(\Gamma)}{\Gamma \vdash \lambda x.e : T \to U \rightsquigarrow \lambda x.\widehat{e}}\ [\text{T-Abs}] \qquad \frac{un(\Gamma)}{\Gamma, \alpha : T \vdash \alpha : T \rightsquigarrow \alpha}\ [\text{T-Id}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \multimap U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2 \vdash e_2 : T \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : U \rightsquigarrow \widehat{e}_1\ \widehat{e}_2}\ [\text{T-App}]$$

$$\frac{\Gamma_1 \vdash e : \&\langle l_i : T_i \rangle_{i \in I} \rightsquigarrow \widehat{e} \quad \forall_{i \in I}(\Gamma_2 \vdash e_i : T_i \multimap T \rightsquigarrow \widehat{e}_i)}{\Gamma_1 + \Gamma_2 \vdash \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} : T \rightsquigarrow \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e}_i\}_{i \in I}}\ [\text{T-Case}]$$

$$\frac{\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e}}{\Gamma \vdash \lambda x.e : T \multimap U \rightsquigarrow \lambda x.\widehat{e}}\ [\text{T-AbsL}] \qquad \frac{un(\Gamma)}{\Gamma, y : T \vdash \wr : T \rightsquigarrow y}\ [\text{T-Query}]$$

$$\frac{\Gamma \vdash e : \oplus\langle l_i : T_i \rangle_{i \in I} \rightsquigarrow \widehat{e} \quad j \in I}{\Gamma \vdash \texttt{select}\ l_j\ e : T_j \rightsquigarrow \texttt{select}\ l_j\ \widehat{e}}\ [\text{T-Select}]$$

$$\frac{\Gamma_1 \vdash e : T \wr\!\multimap U \rightsquigarrow \widehat{e} \quad \Gamma_2 \vdash \wr : T \rightsquigarrow y}{\Gamma_1 + \Gamma_2 \vdash e : U \rightsquigarrow \widehat{e}\ y}\ [\text{T-AppI}]$$

$$\frac{\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e} \quad y\ \text{fresh} \quad un(\Gamma)}{\Gamma \vdash e : T \wr\!\to U \rightsquigarrow \lambda y.\widehat{e}}\ [\text{T-AbsI}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2, x : T, y : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2 \quad y\ \text{fresh}}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, \wr = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2}\ [\text{T-SplitI}]$$

$$\frac{un(\Gamma)}{\Gamma \vdash \texttt{implicit receive} : ?^{\wr}T.S \to T \otimes S \rightsquigarrow \texttt{receive}}\ [\text{T-InI}]$$

$$\frac{\Gamma_1 \vdash \wr : T \rightsquigarrow y \quad \Gamma_2 \vdash e : !^{\wr}T.S \rightsquigarrow \widehat{e}}{\Gamma_1 + \Gamma_2 \vdash e : S \rightsquigarrow \texttt{send}\ y\ \widehat{e}}\ [\text{T-OutI}]$$

$$\frac{\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e} \quad y\ \text{fresh}}{\Gamma \vdash e : T \wr\!\multimap U \rightsquigarrow \lambda y.\widehat{e}}\ [\text{T-AbsLI}]$$

FIGURE 3.14: Typing and translation rules for IM expressions

$$\frac{\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}}{\Gamma \vdash l\vec{b} : l\vec{B} \rightsquigarrow \widehat{l\vec{b}}}\ [\text{T-SEQL}] \qquad \frac{un(\Gamma)}{\Gamma \vdash \epsilon : \epsilon \rightsquigarrow \epsilon}\ [\text{T-EMPTY}]$$

$$\frac{\Gamma_1 \vdash v : T \rightsquigarrow \widehat{v} \quad \Gamma_2 \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}}{\Gamma_1 + \Gamma_2 \vdash v\vec{b} : T\vec{B} \rightsquigarrow \widehat{v\vec{b}}}\ [\text{T-SEQV}]$$

FIGURE 3.15: Typing and translation rules for IM buffer contents

the same way as [T-CONST]. [T-OUTI] translates implicit outputs by inserting a `send` action into the process. The argument for the `send` is a variable from the implicit scope, which we get from the first premise by translating $\wr$ with (a subset of) the input environment. This yields an implicit variable with the appropriate type whilst also satisfying any linearity constraints. Note that [T-OUTI] is the only rule adding outputs directly.

Typing and translation of buffer contents

Typing judgements for buffers follow the same form as typing judgements for expressions. We write $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$ in this case. The translation of buffers can be found in figure 3.15.

Typing and translation of configurations

Typing judgements for configurations (Figure 3.16) follow a slightly different form to those for buffer contents and expressions. We write $\Gamma \vdash C \rhd \Delta \rightsquigarrow \widehat{C}$. This can be read as "under assumptions Γ, the configuration C yields buffer types Δ and is translated as $\widehat{C}$". We define buffer type maps Δ below in Definition 2. The rules [T-THREAD], [T-BUFFER] and [T-NEW] are as in [Gay and Vasconcelos, 2010], augmented with homomorphic translations. The rule [T-PAR] [2] is also similar to its equivalent rule in [Gay and Vasconcelos, 2010], but also contains two new premises. The first computes the buffer types in the configuration $C_1 \parallel C_2$, which are used in the second premise to perform *implicit resolution*. The judgements used in these premises are explained below.

DEFINITION 2. *Buffer types* are triples of the form $(d, n, \vec{B})$. We let Δ range over partial finite maps from channel names to *buffer types* in C. Writing $\Delta + \Delta'$ means that the domains of Δ and Δ' are disjoint.

[2] Note that the rule [T-PAR] of [Gay and Vasconcelos, 2010] uses a *compatibility* relation $S \asymp S'$, which holds exactly when $\overline{S} <: S'$. In this presentation we opt simply to write $\overline{S} <: S'$.

$$\frac{\Gamma \vdash e : T \rightsquigarrow \widehat{e} \quad un(T)}{\Gamma \vdash \langle e \rangle \triangleright \emptyset \rightsquigarrow \langle \widehat{e} \rangle} \quad [\text{T-THREAD}]$$

$$\frac{\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}} \quad |\vec{b}| \leq n}{\Gamma \vdash c \mapsto (d, n, \vec{b}) \triangleright c : (d, n, \vec{B}) \rightsquigarrow c \mapsto (d, n, \widehat{\vec{b}})} \quad [\text{T-BUFFER}]$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma'_1 + \Gamma'_2 \quad \Gamma'_1 \vdash C_1 \triangleright \Delta_1 \rightsquigarrow \widehat{C}_1 \quad \Gamma'_2 \vdash C_2 \triangleright \Delta_2 \rightsquigarrow \widehat{C}_2 \quad \Delta' = \Delta_1 + \Delta_2 \\ \forall c \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B}) \Rightarrow (\vec{B}\ mat\ \Gamma'(c)\ and\ bound(\Gamma'(c)) \leq n)) \\ \forall c, d \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B})\ and\ \Delta'(d) = (c, n', \vec{B}') \Rightarrow \Gamma'(c)/\vec{B} <: \overline{\Gamma'(d)/\vec{B}'})\end{array}}{\Gamma \vdash C_1 \parallel C_2 \triangleright \Delta' \rightsquigarrow \widehat{C}_1 \parallel \widehat{C}_2} \quad [\text{T-PAR}]$$

$$\frac{\Gamma + c_1 : S_1 + c_2 : S_2 \vdash C \triangleright \Delta + c_1 : (c_2, n_1, \vec{B}_1) + c_2 : (c_1, n_2, \vec{B}_2) \rightsquigarrow \widehat{C}}{\Gamma \vdash (\nu c_1 c_2) C \triangleright \Delta \rightsquigarrow (\nu c_1 c_2) \widehat{C}} \quad [\text{T-NEW}]$$

FIGURE 3.16: Typing and translation rules for IM configurations

DEFINITION 3. We define a partial operation of addition on environments:

$$\Gamma + x : T = \begin{cases} \Gamma, x : T & x \notin \mathsf{dom}(\Gamma) \\ \Gamma & \Gamma(x) = T, \mathsf{un}(\Gamma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We extend this to $\Gamma + \Gamma'$ inductively from the base case.

3.5.1 Sources of ambiguity

There are two sources of ambiguity in implicit resolution. The first is in the selection of the implicit variable chosen by the rule [T-QUERY]. We do not specify which variable in the implicit scope should replace a ≀. A possible way to resolve this is to use nesting. Such a solution would select the innermost implicit variable of the appropriate type as the translation for ≀. The Scala compiler uses a more complex version of this strategy, augmented with other selection criteria [Odersky et al., 2018].

The second source of ambiguity results from the insertion of output actions when resolving implicit messages. When a pair of composed processes are resolved, we do not specify which is resolved first. As a result, adjacent implicit inputs can be resolved in multiple ways. Consider the processes:

```
⟨ let p =
    let ≀ = ...
    let c = accept x in
    let n, ≀ = implicit receive in c
```

```
in p ⟩ ‖ ⟨ let q =
    let ≀ = ...
    let d = request x in
    let n, ≀ = implicit receive in d
in q ⟩
```

Implicit resolution should insert two output actions here, one in `p` and the other in `q`. If we resolve `p` before `q`, we obtain the processes:

```
⟨ let p =
    let y = ...
    let c = accept x in
    send y c
    let n, c = receive in c
in p ⟩ ‖ ⟨ let q =
    let y = ...
    let d = request x in
    let n, d = receive in d
    send y d
in q ⟩
```

We could also resolve `q` first and obtain the processes:

```
⟨ let p =
    let y = ...
    let c = accept x in
    let n, c = receive in c
    send y c
in p ⟩ ‖ ⟨ let q =
    let y = ...
    let d = request x in
    send y d
    let n, d = receive in d
in q ⟩
```

We term this situation the *adjacent implicit messages problem*.

As with ambiguity caused by resolution of $\wr$, an implementation could use a simple heuristic such as to resolve the left hand side of parallel composition before the right. In the absence of a proof, it is unclear whether such a heuristic could work in all possible cases of adjacent implicit messages. We leave these as future work.

3.6 Runtime safety of IM

We prove safety of IM's translation into LAST: we show that if we can derive $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$ in IM, then $\widehat{C}$ can be typed suitably in LAST. In order to make this precise, we define a function $(\cdot)^*$, that translates IM's types to standard LAST types. This translation simply erases occurrences of $\wr$, yielding non-implicit analogues of implicit types.

DEFINITION 4 (Translation of types)**.** The definition of $(\cdot)^*$ for session types is given in figure 3.17.

$$
\begin{array}{rclcrcl}
(T \wr\!\!\rightarrow T')^* & = & T^* \to T'^* & \qquad & \texttt{Unit}^* & = & \texttt{Unit} \\
(T \wr\!\!\multimap T')^* & = & T^* \multimap T'^* & & \texttt{end}^* & = & \texttt{end} \\
(T \to T')^* & = & T^* \to T'^* & & (?T.S)^* & = & ?T^*.S^* \\
(T \multimap T')^* & = & T^* \multimap T'^* & & (!T.S)^* & = & !T^*.S^* \\
\&\langle l_i : S_i \rangle_{i\in I}^* & = & \&\langle l_i : S_i^* \rangle_{i\in I} & & (\langle S \rangle^{\mathtt{r}})^* & = & \langle S^* \rangle^{\mathtt{r}} \\
\oplus\langle l_i : S_i \rangle_{i\in I}^* & = & \oplus\langle l_i : S_i^* \rangle_{i\in I} & & (\langle S \rangle^{\mathtt{a}})^* & = & \langle S^* \rangle^{\mathtt{a}} \\
\langle S, S' \rangle^* & = & \langle S^*, S'^* \rangle & & (?^{\flat}T.S)^* & = & ?T^*.S^* \\
(T \otimes T')^* & = & T^* \otimes T'^* & & (!^{\flat}T.S)^* & = & !T^*.S^* \\
(\mu X.S)^* & = & \mu X.S^* & & X^* & = & X
\end{array}
$$

FIGURE 3.17: Translation of IM session types

We extend the definition of $(\cdot)^*$ to buffer types and environments:

DEFINITION 5 (Translation of buffer types and environments)**.** The definition of $(\cdot)^*$ for buffer types is given in figure 3.18. This is lifted pointwise to environments (i.e. $(\Gamma, x : T)^* = \Gamma^*, x : T^*$).

$$
\begin{array}{rclcrcl}
\epsilon^* & = & \epsilon & \qquad & (T\vec{B})^* & = & T^*\vec{B}^* \\
(l\vec{B})^* & = & l\vec{B}^* & & (c, n, \vec{B})^* & = & (c, n, \vec{B}^*)
\end{array}
$$

FIGURE 3.18: Translation of IM buffer types

We call a configuration *fully buffered* if whenever it contains $c \mapsto (c', n, \vec{b})$ then it also contains $c' \mapsto (c, n', \vec{b}')$. We recall the following theorem from [Gay and Vasconcelos, 2010], defining and proving LAST's runtime safety.

THEOREM (Runtime safety of LAST)**.** Let $\Gamma \vdash_{\text{LAST}} C \rhd \Delta$ be a fully buffered LAST configuration, and assume that $C \longrightarrow^* C'$. If C'' is a blocked thread in C', then one if the following applies:

- C'' is $\langle v \rangle$ or $\langle \texttt{send}\ v \rangle$ or $\langle \texttt{request}\ n\ v \rangle$ or $\langle \texttt{accept}\ n\ v \rangle$;
- C'' is $\langle E[\texttt{receive}\ c] \rangle$ and $c \mapsto (\_, \_, \epsilon) \in C'$;
- C'' is $\langle E[\texttt{case}\ c\ \texttt{of}\ \{l_i : e_i\}_{i\in I}] \rangle$ and $c \mapsto (\_, \_, \epsilon) \in C'$.

This result was established in [Gay and Vasconcelos, 2010]. We now state our main result.

THEOREM 1 (Runtime safety of IM)**.** If $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$ is a fully buffered IM configuration, then $\Gamma^* \vdash_{\text{LAST}} \widehat{C} \triangleright \Delta^*$ is a runtime-safe LAST configuration.

Proof. Immediately for Theorem 4. □

We now proceed to prove supporting lemmas and theorems, and theorem 4 itself.

LEMMA 1 (Preservation of membership with $(\cdot)^*$ on environments)**.** $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma^*)$

Proof. Trivially by induction on Definition 5. □

LEMMA 2 (Distributivity of $(\cdot)^*$ over $+$ on environments)**.** If $\Gamma_1 + \Gamma_2$ is defined, then $\Gamma_1^* + \Gamma_2^* = (\Gamma_1 + \Gamma_2)^*$

Proof. By induction on the definition of $+$.
There are two cases where $\Gamma + x : T$ is defined:

- if $x \notin dom(\Gamma)$, then $\Gamma + x : T = \Gamma, x : T$
  - Starting with $(\Gamma, x : T)^*$, by the Definition 5, we get $\Gamma^*, x : T^*$
  - Starting with $\Gamma^* + (x : T)^*$, by the Definition 5, we get $\Gamma^* + x : T^*$
    - Then, by the definition of $+$, we get $\Gamma^*, x : T^*$
- if $x : T \in \Gamma$ and $un(\Gamma)$, then $\Gamma + x : T = \Gamma$
  - Starting with $(\Gamma + x : T)^*$, we get Γ^* by the definition of $+$.
  - Starting with $\Gamma^* + (x : T)^*$,
    - By Definition 5, $\Gamma^* + x : T^*$
    - Then by Lemma 1, Γ^*.

Addition is extended inductively to a partial binary operation on environments, and distributivity therefore holds by the induction hypothesis. □

LEMMA 3 (Preservation of contractivity and closure of types under translation)**.** If $T \in \mathscr{T}$, then $T^* \in \mathscr{T}_{LAST}$. Equally, if $S \in \mathscr{S}$, then $S^* \in \mathscr{S}_{LAST}$.

Proof. From Definition 4 we see that none of the translations change the number or position of type constructors in a type, and therefore contractivity is preserved. We also see that the names in μ-binders and of type variables are not modified by translation and thus closure is preserved. These can be shown formally by a routine induction on T, S. □

LEMMA 4 (Preservation of subtyping with $(\cdot)^*$ on types)**.** If $T <: U$, then $T^* <:_{LAST} U^*$

Proof. We define $\mathcal{R} = \{(T^*, T'^*) \mid (T, T') \in \nu F\}$ and show that $\mathcal{R}$ is a pre-fixpoint of F_{LAST}, the monotone function [Gay and Vasconcelos, 2010] uses to define $<:_{LAST}$. In other words, we show that $\mathcal{R} \subseteq F_{LAST}(\mathcal{R})$. We proceed by induction on $(T, T') \in \nu F$.

- **Case** $(\mathtt{end}, \mathtt{end})$
  - To show: $(\mathtt{end}^*, \mathtt{end}^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\mathtt{end}, \mathtt{end}) \in \nu F_{LAST}$
  - The goal follows immediately from the definition of F_{LAST}.
- **Case** $(?T.S, ?T'.S')$
  - To show: $((?T.S)^*, (?T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(?T^*.S^*, ?T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(!T.S, !T'.S')$
  - To show: $((!T.S)^*, (!T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(!T^*.S^*, !T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(?^{\wr}T.S, ?^{\wr}T'.S')$
  - To show: $((?^{\wr}T.S)^*, (?^{\wr}T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(?T^*.S^*, ?T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(!^{\wr}T.S, !^{\wr}T'.S')$
  - To show: $((!^{\wr}T.S)^*, (!^{\wr}T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(!T^*.S^*, !T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$

  - The goal then follows from the definition of F_{LAST}.

- **Case** $(\&\langle l_i : S_i\rangle_{i\in I}, \&\langle l_j : S'_j\rangle_{j\in J})$
  - To show: $((\&\langle l_i : S_i\rangle_{i\in I})^*, (\&\langle l_j : S'_j\rangle_{j\in J})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\&\langle l_i : S^*_i\rangle_{i\in I}, \&\langle l_j : S'^*_j\rangle_{j\in J}) \in \nu F_{LAST}$
  - By the definition of F, $I \subseteq J, (S_i, S'_i) \in \nu F, \forall i \in I$.
    - Then by the induction hypothesis, $(S^*_i, S'^*_i) \in \nu F_{LAST}, \forall i \in I$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(\oplus\langle l_i : S_i\rangle_{i\in I}, \oplus\langle l_j : S'_j\rangle_{j\in J})$
  - To show: $((\oplus\langle l_i : S_i\rangle_{i\in I})^*, (\oplus\langle l_j : S'_j\rangle_{j\in J})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\oplus\langle l_i : S^*_i\rangle_{i\in I}, \oplus\langle l_j : S'^*_j\rangle_{j\in J}) \in \nu F_{LAST}$
  - By the definition of F, $I \subseteq J, (S_i, S'_i) \in \nu F, \forall i \in I$.
    - Then by the induction hypothesis, $(S^*_i, S'^*_i) \in \nu F_{LAST}, \forall i \in I$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(\langle S, S'\rangle, \langle S\rangle^{\mathtt{a}})$
  - To show: $(\langle S, S'\rangle^*, (\langle S\rangle^{\mathtt{a}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^*, S'^*\rangle, \langle S^*\rangle^{\mathtt{a}}) \in \nu F_{LAST}$
  - By the definition of F, $S, S' \in \mathscr{S}$.
    - Then by Lemma 3, $S^*, S'^* \in \mathscr{S}_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(\langle S, S'\rangle, \langle S'\rangle^{\mathtt{r}})$
  - To show: $(\langle S, S'\rangle^*, (\langle S'\rangle^{\mathtt{r}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^*, S'^*\rangle, \langle S'^*\rangle^{\mathtt{r}}) \in \nu F_{LAST}$
  - By the definition of F, $S, S' \in \mathscr{S}$.
    - Then by Lemma 3, $S^*, S'^* \in \mathscr{S}_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(\langle S\rangle^{\mathtt{a}}, \langle S'\rangle^{\mathtt{a}})$
  - To show: $((\langle S\rangle^{\mathtt{a}})^*, (\langle S'\rangle^{\mathtt{a}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^*\rangle^{\mathtt{a}}, \langle S'^*\rangle^{\mathtt{a}}) \in \nu F_{LAST}$
  - By the definition of F, $(S, S') \in \nu F$.
    - Then by the induction hypothesis, $(S^*, S'^*) \in \nu F_{LAST}$.

  - The goal then follows from the definition of F_{LAST}.

- **Case** $(\langle S \rangle^{\mathtt{r}}, \langle S' \rangle^{\mathtt{r}})$
  - To show: $((\langle S \rangle^{\mathtt{r}})^*, (\langle S' \rangle^{\mathtt{r}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^* \rangle^{\mathtt{r}}, \langle S'^* \rangle^{\mathtt{r}}) \in \nu F_{LAST}$
  - By the definition of F, $(S, S') \in \nu F$.
    - Then by the induction hypothesis, $(S^*, S'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(\langle S_1, S_1' \rangle, \langle S_2, S_2' \rangle)$
  - To show: $(\langle S_1, S_1' \rangle^*, \langle S_2, S_2' \rangle^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S_1^*, S_1'^* \rangle, \langle S_2^*, S_2'^* \rangle) \in \nu F_{LAST}$
  - By the definition of F, $(S_1, S_1'), (S_2, S_2') \in \nu F$.
    - Then by the induction hypothesis, $(S_1^*, S_1'^*), (S_2^*, S_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T \to T', T \multimap T')$
  - To show: $((T \to T')^*, (T \multimap T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \to T'^*, T^* \multimap T'^*) \in \nu F_{LAST}$
  - By the definition of F, $T, T' \in \mathscr{T}$.
    - Then by Lemma 3, $T^*, T'^* \in \mathscr{T}_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T_1 \to T_1', T_2 \to T_2')$
  - To show: $((T_1 \to T_1')^*, (T_2 \to T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \to T_1'^*, T_2^* \to T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T_1 \multimap T_1', T_2 \multimap T_2')$
  - To show: $((T_1 \multimap T_1')^*, (T_2 \multimap T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \multimap T_1'^*, T_2^* \multimap T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.

  - The goal then follows from the definition of F_{LAST}.

- **Case** $(\mu X.S, S')$
  - To show: $((\mu X.S)^*, S'^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\mu X.S^*, S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(S[^{\mu X.S}/_X], S') \in \nu F$.
    - Then by the induction hypothesis, $(S^*[^{\mu X.S^*}/_X], S'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(S, \mu X.S')$
  - To show: $(S^*, (\mu X.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(S^*, \mu X.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(S, S'[^{\mu X.S'}/_X]) \in \nu F$.
    - Then by the induction hypothesis, $(S^*, S'^*[^{\mu X.S'^*}/_X]) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T_1 \wr\!\!\rightarrow T_1', T_2 \wr\!\!\rightarrow T_2')$
  - To show: $((T_1 \wr\!\!\rightarrow T_1')^*, (T_2 \wr\!\!\rightarrow T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \rightarrow T_1'^*, T_2^* \rightarrow T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T_1 \wr\!\!\multimap T_1', T_2 \wr\!\!\multimap T_2')$
  - To show: $((T_1 \wr\!\!\multimap T_1')^*, (T_2 \wr\!\!\multimap T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \multimap T_1'^*, T_2^* \multimap T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T \wr\!\!\rightarrow T', T \wr\!\!\multimap T')$
  - To show: $((T \wr\!\!\rightarrow T')^*, (T \wr\!\!\multimap T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \rightarrow T'^*, T^* \multimap T'^*) \in \nu F_{LAST}$
  - By the definition of F, $T, T' \in \mathscr{T}$.
    - Then by Lemma 3, $T^*, T'^* \in \mathscr{T}_{LAST}$.

  - The goal then follows from the definition of F_{LAST}.

- **Case** $(T \to T', T \wr\to T')$
  - To show: $((T \to T')^*, (T \wr\to T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \to T'^*, T^* \to T'^*) \in \nu F_{LAST}$
  - The goal holds immediately by reflexivity of subtyping in LAST [Gay and Vasconcelos, 2010].

- **Case** $(T \multimap T', T \wr\multimap T')$
  - To show: $((T \multimap T')^*, (T \wr\multimap T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \multimap T'^*, T^* \multimap T'^*) \in \nu F_{LAST}$
  - The goal holds immediately by reflexivity of subtyping in LAST [Gay and Vasconcelos, 2010].

- **Case** $(!T.S, !^{\wr}T.S)$
  - To show: $((!T.S)^*, (!^{\wr}T.S)^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(!T^*.S^*, !T^*.S^*) \in \nu F_{LAST}$
  - The goal holds immediately by reflexivity of subtyping in LAST [Gay and Vasconcelos, 2010].

□

Lemma 5 (Preservation of linearity and nonlinearity with $(\cdot)^*$ on types)**.** Let $T' = T^*$.

- $un(T) \iff un(T')$.
- $lin(T) \iff lin(T')$.

Proof. Trivially by induction on Definition 4. All linear types translate to linear types, and all unlimited types translate to unlimited types. □

Lemma 6 (Preservation of linearity and nonlinearity with $(\cdot)^*$ on environments)**.** Let $\Gamma' = \Gamma^*$.

- $un(\Gamma) \iff un(\Gamma')$.
- $lin(\Gamma) \iff lin(\Gamma')$.

Proof. Trivially by induction on Definition 5 and Lemma 5. □

Lemma 7 (Type-preserving translation of type schemes for constants)**.** For each constant type scheme $k : T$, there is a type scheme in LAST of the form $k : T^*$.

Proof. Follows immediately from Definition 4. □

THEOREM 2 (Type-preserving translation of expressions)**.** Let e be an expression with implicits, let Γ be an environment that may contain implicit types, and let T be a type that may contain implicit types. If $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$, then $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$.

Proof. By induction on $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$

- **Case** $\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : T \otimes U \rightsquigarrow (\widehat{e_1}, \widehat{e_2})$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} (\widehat{e_1}, \widehat{e_2}) : (T \otimes U)^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} (\widehat{e_1}, \widehat{e_2}) : (T \otimes U)^*$
    - or by Definition 4, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} (\widehat{e_1}, \widehat{e_2}) : T^* \otimes U^*$
  - By inversion of **T-Pair**:
    - $\Gamma_1 \vdash e_1 : T \rightsquigarrow \widehat{e_1}$
    - $\Gamma_2 \vdash e_2 : U \rightsquigarrow \widehat{e_2}$
  - By the induction hypothesis:
    - $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e_1} : T^*$
    - $\Gamma_2^* \vdash_{\text{LAST}} \widehat{e_2} : U^*$
  - By $\textbf{T-Pair}_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} (\widehat{e_1}, \widehat{e_2}) : T^* \otimes U^*$
- **Case** $\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : U \rightsquigarrow \widehat{e_1}\ \widehat{e_2}$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{e_1}\ \widehat{e_2} : U^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e_1}\ \widehat{e_2} : U^*$
  - By inversion of **T-App**,
    - $\Gamma_2 \vdash e_2 : T \rightsquigarrow \widehat{e_2}$
    - $\Gamma_1 \vdash e_1 : T \multimap U \rightsquigarrow \widehat{e_1}$
  - By the induction hypothesis:
    - $\Gamma_2^* \vdash_{\text{LAST}} \widehat{e_2} : T^*$
    - $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e_1} : (T \multimap U)^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e_1} : T^* \multimap U^*$
  - By $\textbf{T-App}_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e_1}\ \widehat{e_2} : U^*$
- **Case** $\Gamma \vdash \lambda x.e : T \to U \rightsquigarrow \lambda x.\widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : (T \to U)^*$
    - or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \to U^*$
  - By inversion of **T-Abs**:

  - $un(\Gamma)$
    - and by Lemma 6, $un(\Gamma^*)$
  - $\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e}$
    - and by the induction hypothesis, $(\Gamma, x : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
    - then by Definition 5, $\Gamma^*, x : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - By **T-Abs**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \to U^*$

- **Case** $\Gamma \vdash \lambda x.e : T \multimap U \rightsquigarrow \lambda x.\widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : (T \multimap U)^*$
    - or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \multimap U^*$
  - By inversion of **T-AbsL**:
    - $\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $(\Gamma, x : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
      - then by Definition 5, $\Gamma^*, x : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - By **T-AbsL**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \multimap U^*$

- **Case** $\Gamma, \alpha : T \vdash \alpha : T \rightsquigarrow \alpha$
  - To show: $(\Gamma, \alpha : T)^* \vdash_{\text{LAST}} \alpha : T^*$
    - or by Definition 5, $\Gamma^*, \alpha : T^* \vdash_{\text{LAST}} \alpha : T^*$
  - By inversion of **T-ID**, $un(\Gamma)$
    - and by Lemma 6, $un(\Gamma^*)$
  - By **T-ID**$_{GV}$, $\Gamma^*, \alpha : T^* \vdash_{\text{LAST}} \alpha : T^*$

- **Case** $\Gamma \vdash k : T \rightsquigarrow k$
  - To show: $\Gamma^* \vdash_{\text{LAST}} k : T^*$
  - By inversion of **T-Const**, we have:
    - $un(\Gamma)$
      - and by Lemma 6, $un(\Gamma^*)$
    - $k : T$
      - and by Lemma 7, $k : T^*$
  - By **T-ID**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} k : T^*$

- **Case** $\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, y = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x, y = \widehat{e_1}\ \texttt{in}\ \widehat{e_2}$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e_1}\ \texttt{in}\ \widehat{e_2} : V^*$

  - Or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e_1}\ \texttt{in}\ \widehat{e_2} : V^*$
- By inversion of **T-Split**, we have:
  - $\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e_1}$
    - and by the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e_1} : (T \otimes U)^*$
    - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e_1} : T^* \otimes U^*$
  - $\Gamma_2, x : T, y : U \vdash e_2 : V \rightsquigarrow \widehat{e_2}$
    - and by the induction hypothesis, $(\Gamma_2, x : T, y : U)^* \vdash_{\text{LAST}} \widehat{e_2} : V^*$
    - then by Definition 5, $\Gamma_2^*, x : T^*, y : U^* \vdash_{\text{LAST}} \widehat{e_2} : V^*$
  - By **T-Split**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e_1}\ \texttt{in}\ \widehat{e_2} : V^*$

- **Case** $\Gamma \vdash \texttt{select}\ l_j\ e : T_j \rightsquigarrow \texttt{select}\ l_j\ \widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \texttt{select}\ l_j\ \widehat{e} : T_j^*$
  - By inversion of **T-Select**, we have:
    - $\Gamma \vdash e : \oplus\langle l_i : T_i\rangle_{i \in I} \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : \oplus\langle l_i : T_i\rangle_{i \in I}^*$
      - then by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : \oplus\langle l_i : T_i^*\rangle_{i \in I}$
    - $j \in I$
  - By **T-Select**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \texttt{select}\ l_j\ \widehat{e} : T_j^*$

- **Case** $\Gamma_1 + \Gamma_2 \vdash \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} : T \rightsquigarrow \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e_i}\}_{i \in I}$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e_i}\}_{i \in I} : T^*$
    - Or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e_i}\}_{i \in I} : T^*$
  - By inversion of **T-Case**, we have:
    - $\Gamma_1 \vdash e : \&\langle l_i : T_i\rangle_{i \in I} \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : \&\langle l_i : T_i\rangle_{i \in I}^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : \&\langle l_i : T_i^*\rangle_{i \in I}$
    - $\forall i \in I.\quad \Gamma_2 \vdash e_i : T_i \multimap T \rightsquigarrow \widehat{e_i}$
      - and by the induction hypothesis, $\forall i \in I.\quad \Gamma_2^* \vdash_{\text{LAST}} \widehat{e_i} : (T_i \multimap T)^*$
      - then by Definition 4, $\forall i \in I.\quad \Gamma_2^* \vdash_{\text{LAST}} \widehat{e_i} : T_i^* \multimap T^*$
  - By **T-Case**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e_i}\}_{i \in I} : T^*$

- **Case** $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$
  - By inversion of **T-Sub**, we have:

  - $\Gamma \vdash e : U \rightsquigarrow \widehat{e}$
    - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - $T <: U$
    - and by Lemma 4, $T^* <: U^*$
  - Then by **T-Sub**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$

- **Case** $\Gamma, y : T \vdash \wr : T \rightsquigarrow y$
  - To show: $(\Gamma, y : T)^* \vdash_{\text{LAST}} y : T^*$
    - or by Definition 5, $\Gamma^*, y : T^* \vdash_{\text{LAST}} y : T^*$
  - By inversion of **T-Query**, $un(\Gamma)$
    - and by Lemma 6, $un(\Gamma^*)$
  - By **T-ID**$_{GV}$, $\Gamma^*, y : T^* \vdash_{\text{LAST}} y : T^*$

- **Case** $\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, \wr = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$
    - Or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$
  - By inversion of **T-SplitI**, we have:
    - y *fresh*
    - $\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1$
      - and by the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : (T \otimes U)^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : T^* \otimes U^*$
    - $\Gamma_2, x : T, y : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2$
      - and by the induction hypothesis, $(\Gamma_2, x : T, y : U)^* \vdash_{\text{LAST}} \widehat{e}_2 : V^*$
      - then by Definition 5, $\Gamma_2^*, x : T^*, y : U^* \vdash_{\text{LAST}} \widehat{e}_2 : V^*$
  - By **T-Split**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$

- **Case** $\Gamma_1 + \Gamma_2 \vdash e : U \rightsquigarrow \widehat{e}\, y$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{e}\, y : U^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}\, y : U^*$
  - By inversion of **T-AppI**,
    - $\Gamma_1 \vdash e : T \wr\!\multimap U \rightsquigarrow \widehat{e}$
      - By the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : (T \wr\!\multimap U)^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : T^* \multimap U^*$
    - $\Gamma_2 \vdash \wr : T \rightsquigarrow y$

- By the induction hypothesis, $\Gamma_2^* \vdash_{\text{LAST}} y : T^*$

– By **T-App**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}\, y : U^*$

- **Case** $\Gamma \vdash e : T \wr\!\!\rightarrow U \rightsquigarrow \lambda y.\widehat{e}$

– To show: $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : (T \wr\!\!\rightarrow U)^*$

- or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \to U^*$

– By inversion of **T-AbsI**:

- y *fresh*
- $un(\Gamma)$
  - and by Lemma 6, $un(\Gamma^*)$
- $\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e}$
  - and by the induction hypothesis, $(\Gamma, y : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - then by Definition 5, $\Gamma^*, y : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$

– By **T-Abs**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \to U^*$

- **Case** $\Gamma \vdash e : T \wr\!\!\multimap U \rightsquigarrow \lambda y.\widehat{e}$

– To show: $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : (T \wr\!\!\multimap U)^*$

- or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \multimap U^*$

– By inversion of **T-AbsI**:

- y *fresh*
- $\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e}$
  - and by the induction hypothesis, $(\Gamma, y : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - then by Definition 5, $\Gamma^*, y : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$

– By **T-AbsL**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \multimap U^*$

- **Case** $\Gamma \vdash \texttt{implicit receive} : ?^{\wr} T.S \to T \otimes S \rightsquigarrow \texttt{receive}$

– To show: $\Gamma^* \vdash \texttt{receive} : (?^{\wr} T.S \to T \otimes S)^*$

- or, by Definition 4, $\Gamma^* \vdash \texttt{receive} : ?T^*.S^* \to T^* \otimes S^*$

– By inversion of **T-InI**, $un(\Gamma)$

- and by Lemma 6, $un(\Gamma^*)$

– Result follows immediately from the type schema for `receive` and **T-Const**$_{GV}$.

- **Case** $\Gamma_1 + \Gamma_2 \vdash e : S \rightsquigarrow \texttt{send}\, y\, \widehat{e}$

– To show: $(\Gamma_1 + \Gamma_2)^* \vdash \texttt{send}\, y\, \widehat{e} : S^*$

- By inversion of **T-OutI**,
  - $\Gamma_1 \vdash \wr : T \rightsquigarrow y$
    - and by the induction hypothesis, $\Gamma_1^* \vdash y : T^*$
  - $\Gamma_2 \vdash e : !^{\wr}T.S \rightsquigarrow \widehat{e}$
    - and by the induction hypothesis, $\Gamma_2^* \vdash \widehat{e} : (!^{\wr}T.S)^*$
    - Then by Definition 4, $\Gamma_2^* \vdash \widehat{e} : !T^*.S^*$
- **Case** $un(T)$:
  - By **T-App**$_{GV}$:
    - $\Gamma_1^* + \vdash \texttt{send}\ y : !T^*.S^* \to S^*$
    - and by Definition 5, $\Gamma_1^* \vdash \texttt{send}\ y : !T^*.S^* \to S^*$
  - Again **T-App**$_{GV}$:
    - $\Gamma_1^* + \Gamma_2^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$
    - Then by Lemma 2, $(\Gamma_1 + \Gamma_2)^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$
- **Case** *not* $un(T)$:
  - By **T-App**$_{GV}$:
    - $\Gamma_1^* + \vdash \texttt{send}\ y : !T^*.S^* \multimap S^*$
    - and by Definition 5, $\Gamma_1^* \vdash \texttt{send}\ y : !T^*.S^* \multimap S^*$
  - Again **T-App**$_{GV}$:
    - $\Gamma_1^* + \Gamma_2^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$
    - Then by Lemma 2, $(\Gamma_1 + \Gamma_2)^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$

□

THEOREM 3 (Type and size-preserving translation of buffer contents)**.** Let $\vec{b}$ be a buffer with implicits, let Γ be an environment that may contain implicit types, and let $\vec{B}$ be a type vector that may contain implicit types. If $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$, then $\Gamma^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$.

Proof. By induction on $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$

- **Case** $\Gamma \vdash \epsilon : \epsilon \rightsquigarrow \epsilon$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \epsilon : \epsilon^*$
    - or by Definition 5, $\Gamma^* \vdash_{\text{LAST}} \epsilon : \epsilon$
  - By inversion of **T-Empty**, $un(\Gamma)$
    - and by Lemma 5, $un(\Gamma^*)$
  - By **T-Empty**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \epsilon : \epsilon$

  - Trivially $|\epsilon| = |\epsilon|$

- **Case** $\Gamma_1 + \Gamma_2 \vdash v\vec{b} : T\vec{B} \rightsquigarrow \widehat{v}\widehat{\vec{b}}$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : (T\vec{B})^*$
    - or by Definition 5, $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : T^*\vec{B}^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : T^*\vec{B}^*$
  - By inversion of **T-SeqV**,
    - $\Gamma_1 \vdash v : T \rightsquigarrow \widehat{v}$
      - and by Theorem 2, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{v} : T^*$
    - $\Gamma_2 \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$
      - and by the induction hypothesis, $\Gamma_2^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$
  - By **T-SeqV**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : T^*\vec{B}^*$
  - Since $|\vec{b}| = |\widehat{\vec{b}}|$, $|v| = 1$ and $|\widehat{v}| = 1$, then $|v\vec{b}| = |\widehat{v}\widehat{\vec{b}}|$

- **Case** $\Gamma \vdash l\vec{b} : l\vec{B} \rightsquigarrow l\widehat{\vec{b}}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} l\widehat{\vec{b}} : (l\vec{B})^*$
    - or by Definition 5, $\Gamma^* \vdash_{\text{LAST}} l\widehat{\vec{b}} : l\vec{B}^*$
  - By inversion of **T-SeqL**, $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$
    - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$
  - By **T-SeqL**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} l\widehat{\vec{b}} : l\vec{B}^*$
  - Since $|\vec{b}| = |\widehat{\vec{b}}|$ and $|l| = 1$, then $|l\vec{b}| = |l\widehat{\vec{b}}|$

□

Lemma 8 (Preservation of matching under translation)**.** If $\vec{B}\ \mathit{mat}\ S$ then $\vec{B}^*\ \mathit{mat}\ S^*$

Proof. By induction on $\vec{B}\ \mathit{mat}\ S$.

- **Case** $\epsilon\ \mathit{mat}\ S$
  - To show: $\epsilon\ \mathit{mat}\ S^*$
  - Immediate from **M-Empty**

- **Case** $U\vec{B}\ \mathit{mat}?T.S$
  - To show: $(U\vec{B})^*\ \mathit{mat}\ (?T.S)^*$
    - or by Definition 5, $U^*\vec{B}^*\ \mathit{mat}\ (?T.S)^*$
    - or by Definition 4, $U^*\vec{B}^*\ \mathit{mat}\ ?T^*.S^*$

  - By inversion of **M-Out**,
    - $\vec{B}$ *mat* S
      - and by the induction hypothesis, $\vec{B}^*$ *mat* S^*
    - $U <: T$
      - and by Lemma 4, $U^* <: T^*$
  - By **M-Out**, $U^*\vec{B}^*$ *mat* $?T^*.S^*$

- **Case** $U\vec{B}$ *mat*$?^{\downarrow}T.S$
  - To show: $(U\vec{B})^*$ *mat* $(?^{\downarrow}T.S)^*$
    - or by Definition 5, $U^*\vec{B}^*$ *mat* $(?^{\downarrow}T.S)^*$
    - or by Definition 4, $U^*\vec{B}^*$ *mat* $?T^*.S^*$
  - By inversion of **M-Out**,
    - $\vec{B}$ *mat* S
      - and by the induction hypothesis, $\vec{B}^*$ *mat* S^*
    - $U <: T$
      - and by Lemma 4, $U^* <: T^*$
  - By **M-Out**, $U^*\vec{B}^*$ *mat* $?T^*.S^*$

- **Case** $l\vec{B}$ *mat* $\&\langle ..., l : S, ... \rangle$
  - To show: $(l\vec{B})^*$ *mat* $\&\langle ..., l : S, ... \rangle^*$
    - or by Definition 5, $l\vec{B}^*$ *mat* $\&\langle ..., l : S, ... \rangle^*$
    - or by Definition 4, $l\vec{B}^*$ *mat* $\&\langle ..., l : S^*, ... \rangle$
  - By inversion of **M-Case**, $\vec{B}$ *mat* S
    - and by the induction hypothesis, $\vec{B}^*$ *mat* S^*
  - By **M-Case**, $l\vec{B}^*$ *mat* $\&\langle ..., l : S^*, ... \rangle$

□

LEMMA 9 (Preservation of session type bounds under translation)**.** $bound(S) = bound(S^*)$

Proof. Immediate from the definitions of $bound(\cdot)$ and $bound_{LAST}(\cdot)$ in [Gay and Vasconcelos, 2010]. □

LEMMA 10 (Preservation of duality under translation)**.** For all sessions S, $(\overline{S})^* = \overline{S^*}$.

Proof. By induction on S.

- **Case** $?T.S$

  - To show: $(\overline{?T.S})^* = \overline{(?T.S)^*}$
  - By Definition 4, $(\overline{?T.S})^* = \overline{?T^*.S^*}$
  - By the definition of duality, $(!T.\overline{S})^* = !T^*.\overline{S^*}$
  - By Definition 4, $!T^*.\overline{S}^* = !T^*.\overline{S^*}$
  - By the induction hypothesis, $!T^*.\overline{S^*} = !T^*.\overline{S^*}$

- **Case** $!T.S$
  - To show: $(\overline{!T.S})^* = \overline{(!T.S)^*}$
  - By Definition 4, $(\overline{!T.S})^* = \overline{!T^*.S^*}$
  - By the definition of duality, $(?T.\overline{S})^* = ?T^*.\overline{S^*}$
  - By Definition 4, $?T^*.\overline{S}^* = ?T^*.\overline{S^*}$
  - By the induction hypothesis, $?T^*.\overline{S^*} = ?T^*.\overline{S^*}$

- **Case** $?^{\downarrow}T.S$
  - To show: $(\overline{?^{\downarrow}T.S})^* = \overline{(?^{\downarrow}T.S)^*}$
  - By Definition 4, $(\overline{?^{\downarrow}T.S})^* = \overline{?T^*.S^*}$
  - By the definition of duality, $(!^{\downarrow}T.\overline{S})^* = !T^*.\overline{S^*}$
  - By Definition 4, $!T^*.\overline{S}^* = !T^*.\overline{S^*}$
  - By the induction hypothesis, $!T^*.\overline{S^*} = !T^*.\overline{S^*}$

- **Case** $!^{\downarrow}T.S$
  - To show: $(\overline{!^{\downarrow}T.S})^* = \overline{(!^{\downarrow}T.S)^*}$
  - By Definition 4, $(\overline{!^{\downarrow}T.S})^* = \overline{!T^*.S^*}$
  - By the definition of duality, $(?^{\downarrow}T.\overline{S})^* = ?T^*.\overline{S^*}$
  - By Definition 4, $?T^*.\overline{S}^* = ?T^*.\overline{S^*}$
  - By the induction hypothesis, $?T^*.\overline{S^*} = ?T^*.\overline{S^*}$

- **Case** $\oplus\langle l_i : S_i\rangle_{i\in I}$
  - To show: $\overline{\oplus\langle l_i : S_i\rangle_{i\in I}}^* = \overline{\oplus\langle l_i : S_i\rangle^*_{i\in I}}$
  - By Definition 4, $\overline{\oplus\langle l_i : S_i\rangle_{i\in I}}^* = \overline{\oplus\langle l_i : S_i^*\rangle_{i\in I}}$
  - By the definition of duality, $\&\langle l_i : \overline{S_i}\rangle^*_{i\in I} = \&\langle l_i : \overline{S^*}_i\rangle_{i\in I}$
  - By Definition 4, $\&\langle l_i : \overline{S_i}^*\rangle_{i\in I} = \&\langle l_i : \overline{S^*}_i\rangle_{i\in I}$
  - For all $i \in I$ by the induction hypothesis, $\&\langle l_i : \overline{S^*}_i\rangle_{i\in I} = \&\langle l_i : \overline{S^*}_i\rangle_{i\in I}$

- **Case** $\&\langle l_i : S_i\rangle_{i\in I}$

  - To show: $\overline{\&\langle l_i : S_i\rangle_{i\in I}}^* = \overline{\&\langle l_i : S_i\rangle^*_{i\in I}}$
  - By Definition 4, $\overline{\&\langle l_i : S_i\rangle_{i\in I}}^* = \overline{\&\langle l_i : S_i^*\rangle_{i\in I}}$
  - By the definition of duality, $\oplus\langle l_i : \overline{S}_i\rangle^*_{i\in I} = \oplus\langle l_i : \overline{S^*}_i\rangle_{i\in I}$
  - By Definition 4, $\oplus\langle l_i : \overline{S}_i^*\rangle_{i\in I} = \oplus\langle l_i : \overline{S^*}_i\rangle_{i\in I}$
  - For all $i \in I$ by the induction hypothesis, $\oplus\langle l_i : \overline{S^*}_i\rangle_{i\in I} = \oplus\langle l_i : \overline{S^*}_i\rangle_{i\in I}$

- **Case** $\mathtt{end}$
  - To show: $(\overline{\mathtt{end}})^* = \overline{\mathtt{end}^*}$
  - By Definition 4, $(\overline{\mathtt{end}})^* = \overline{\mathtt{end}}$
  - By the definition of duality, $\mathtt{end}^* = \mathtt{end}$
  - By Definition 4, $\mathtt{end} = \mathtt{end}$

- **Case** X
  - To show: $\overline{X}^* = \overline{X^*}$
  - By Definition 4, $\overline{X}^* = \overline{X}$
  - By the definition of duality, $X^* = X$
  - By Definition 4, $X = X$

- **Case** $\mu X.S$
  - To show: $(\overline{\mu X.S})^* = \overline{(\mu X.S)^*}$
  - By Definition 4, $(\overline{\mu X.S})^* = \overline{\mu X.S^*}$
  - By the definition of duality, $(\mu X.\overline{S})^* = \mu X.\overline{S^*}$
  - By Definition 4, $\mu X.\overline{S}^* = \mu X.\overline{S^*}$
  - By the induction hypothesis, $\mu X.\overline{S^*} = \mu X.\overline{S^*}$

□

Lemma 11 (Preservation of postfix under translation)**.** For all sessions and buffers S, $\vec{B}$, $(S/\vec{B})^* = S^*/\vec{B}^*$

Proof. By induction on S, $\vec{B}$.

- **Case** S, ϵ:
  - To show: $(S/\epsilon)^* = S^*/\epsilon^*$
  - By the definition of postfixes, $S^* = S^*/\epsilon^*$
  - By Definition 5, $S^* = S^*/\epsilon$

  - By the definition of postfixes, $S^* = S^*$

- **Case** $?T.S, U\vec{B}$:
  - To show: $(?T.S/U\vec{B})^* = (?T.S)^*/(U\vec{B})^*$
  - By the definition of postfixes, $(S/\vec{B})^* = (?T.S)^*/(U\vec{B})^*$
  - By Definition 5, $(S/\vec{B})^* = (?T.S)^*/U^*\vec{B}^*$
  - By Definition 4, $(S/\vec{B})^* = ?T^*.S^*/U^*\vec{B}^*$
  - By the definition of postfixes, $(S/\vec{B})^* = S^*/\vec{B}^*$
  - By the induction hypothesis, $(S/\vec{B})^* = (S/\vec{B})^*$

- **Case** $?^\flat T.S, U\vec{B}$:
  - To show: $(?^\flat T.S/U\vec{B})^* = (?^\flat T.S)^*/(U\vec{B})^*$
  - By the definition of postfixes, $(S/\vec{B})^* = (?^\flat T.S)^*/(U\vec{B})^*$
  - By Definition 5, $(S/\vec{B})^* = (?^\flat T.S)^*/U^*\vec{B}^*$
  - By Definition 4, $(S/\vec{B})^* = ?T^*.S^*/U^*\vec{B}^*$
  - By the definition of postfixes, $(S/\vec{B})^* = S^*/\vec{B}^*$
  - By the induction hypothesis, $(S/\vec{B})^* = (S/\vec{B})^*$

- **Case** $\&\langle ..., l : S, ... \rangle, l\vec{B}$:
  - To show: $(\&\langle ..., l : S, ... \rangle/l\vec{B})^* = (\&\langle ..., l : S, ... \rangle)^*/(l\vec{B})^*$
  - By the definition of postfixes, $(S/\vec{B})^* = (\&\langle ..., l : S, ... \rangle)^*/(l\vec{B})^*$
  - By Definition 5, $(S/\vec{B})^* = (\&\langle ..., l : S, ... \rangle)^*/l\vec{B}^*$
  - By Definition 4, $(S/\vec{B})^* = \&\langle ..., l : S^*, ... \rangle/l\vec{B}^*$
  - By the definition of postfixes, $(S/\vec{B})^* = S^*/\vec{B}^*$
  - By the induction hypothesis, $(S/\vec{B})^* = (S/\vec{B})^*$

□

THEOREM 4 (Type-preserving translation of configurations)**.** Let C be an configuration with implicits, and let Γ and Δ be environments that may contain implicit types. If $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$, then $\Gamma^* \vdash_{\text{LAST}} \widehat{C} \triangleright \Delta^*$.

Proof. By induction on $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$

- **Case** $\Gamma \vdash \langle e \rangle \triangleright \emptyset \rightsquigarrow \langle \widehat{e} \rangle$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \langle \widehat{e} \rangle \triangleright *$

  - By inversion of **T-Thread**,
    - $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$
    - $un(T)$
      - and by Lemma 5, $un(T^*)$
  - By **T-Thread**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \langle \widehat{e} \rangle \triangleright {}^*$

- **Case** $\Gamma \vdash c \mapsto (d, n, \vec{b}) \triangleright c : (d, n, \vec{B}) \rightsquigarrow c \mapsto (d, n, \widehat{\vec{b}})$
  - To show: $\Gamma^* \vdash_{\text{LAST}} c \mapsto (d, n, \widehat{\vec{b}}) \triangleright (c : (d, n, \vec{B}))^*$
    - or by Definition 5, $\Gamma^* \vdash_{\text{LAST}} c \mapsto (d, n, \widehat{\vec{b}}) \triangleright c : (d, n, \vec{B}^*)$
  - By inversion of **T-Buffer**, $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$ and $|\vec{b}| \leq n$
  - By Theorem 3, $\Gamma^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$
    - and since $|\vec{b}| \leq n$ and $|\vec{b}| = |\widehat{\vec{b}}|$, $|\widehat{\vec{b}}| \leq n$
  - By Theorem 2, $\Gamma^* \vdash \widehat{\vec{b}} : \vec{B}^*$
  - By **T-Buffer**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} c \mapsto (d, n, \widehat{\vec{b}}) \triangleright c : (d, n, \vec{B}^*)$

- **Case** $\Gamma \vdash C_1 \parallel C_2 \triangleright \Delta \rightsquigarrow \widehat{C}_1 \parallel \widehat{C}_2$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \widehat{C}_1 \parallel \widehat{C}_2 \triangleright \Delta^*$
  - By inversion of **T-Par**,
    - $\Gamma' = \Gamma'_1 + \Gamma'_2$
    - $\Delta' = \Delta_1 + \Delta_2$
    - $\Gamma'_1 \vdash C_1 \triangleright \Delta_1 \rightsquigarrow \widehat{C}_1$
    - $\Gamma'_2 \vdash C_2 \triangleright \Delta_2 \rightsquigarrow \widehat{C}_2$
    - $\forall c \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B}) \Rightarrow (\vec{B}\ mat\ \Gamma'(c)\ and\ bound(\Gamma'(c)) \leq n))$
    - $\forall c, d \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B})\ and\ \Delta'(d) = (c, n', \vec{B}') \Rightarrow \Gamma'(c)/\vec{B} <: \overline{\Gamma'(d)/\vec{B}'})$
  - By the induction hypothesis,
    - $\Gamma'^*_1 \vdash_{\text{LAST}} \widehat{C}_1 \triangleright \Delta^*_1$
    - $\Gamma'^*_2 \vdash_{\text{LAST}} \widehat{C}_2 \triangleright \Delta^*_2$
  - By Lemma 2,
    - $\Gamma'^* = (\Gamma'_1 + \Gamma'_2)^* = \Gamma'^*_1 + \Gamma'^*_2$
  - Trivially we have $\Delta'^* = (\Delta'_1 + \Delta'_2)^* = \Delta'^*_1 + \Delta'^*_2$
  - By Definitions 5 and 4,

  - $c \in dom(\Gamma') \cap dom(\Delta')$ *and* $\Delta'(c) = (d, n, \vec{B}) \Rightarrow$
    $c \in dom(\Gamma'^*) \cap dom(\Delta'^*)$ *and* $\Delta'^*(c) = (d, n, \vec{B}^*)$
  - then by Lemmas 8 and 9,
    $\forall c \in dom(\Gamma'^*) \cap dom(\Delta'^*).(\Delta'^*(c) = (d, n, \vec{B}^*) \Rightarrow$
    $(\vec{B}^*$ *mat* $\Gamma'(c)^*$ *and bound*$(\Gamma'(c)^*) \leq n))$
- From the definitions.
  - $\forall c, d \in dom(\Gamma'^*) \cap dom(\Delta'^*).$
    $(\Delta'^*(c) = (d, n, \vec{B}^*)$ *and*
    $\Delta'^*(d) = (c, n', \vec{B}'^*) \Rightarrow \Gamma'^*(c)/\vec{B}^* \asymp \Gamma'^*(d)/\vec{B}'^*)$
- From the definitions.

- **Case** $\Gamma \vdash (\nu c_1 c_2)C \rhd \Delta \rightsquigarrow (\nu c_1 c_2)\widehat{C}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} (\nu c_1 c_2)\widehat{C} \rhd \Delta^*$
  - By inversion of **T-New**,
    $\Gamma + c_1 : S_1 + c_2 : S_2 \vdash C \rhd \Delta + c_1 : (c_2, n_1, \vec{B}_1) + c_2 : (c_1, n_2, \vec{B}_2) \rightsquigarrow \widehat{C}$
  - By the induction hypothesis,
    $(\Gamma + c_1 : S_1 + c_2 : S_2)^* \vdash \widehat{C} \rhd (\Delta + c_1 : (c_2, n_1, \vec{B}_1) + c_2 : (c_1, n_2, \vec{B}_2))^*$
  - By Definition 5,
    $\Gamma^* + c_1 : S_1^* + c_2 : S_2^* \vdash \widehat{C} \rhd \Delta^* + c_1 : (c_2, n_1, \vec{B}_1)^* + c_2 : (c_1, n_2, \vec{B}_2)^*$
  - By Definition 5,
    $\Gamma^* + c_1 : S_1^* + c_2 : S_2^* \vdash \widehat{C} \rhd \Delta^* + c_1 : (c_2, n_1, \vec{B}_1^*) + c_2 : (c_1, n_2, \vec{B}_2^*)$
  - By **T-New**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} (\nu c_1 c_2)\widehat{C} \rhd \Delta^*$

□

3.7 Conclusion

In this chapter we introduced IM, the first programming language with implicit messages. We have seen that implicit messages facilitate concurrent versions of the typical usage patterns of implicit functions in sequential languages. We have also presented a novel solution to the repeated rebinding problem of linear languages that leverages implicit functions.

IM demonstrates implicit functions and messages in the context of the concurrent functional language LAST, showing the benefits that implicit program constructs offer concurrent programming languages. LAST includes features not necessary for implicit messages, such as lambda abstraction and subtyping. In the next chapter we will study implicit messages in a simpler context: pi calculus. This simpler setting affords a clearer presentation of the idea of implicit messages.

Chapter 4

Pi Calculus with Implicit Messages

4.1 Introduction

This chapter introduces the calculus PIIM, a calculus that adds implicit messages to PLST [Giunti and Vasconcelos, 2013]. This formulation shows that implicit messages are possible without implicit functions, and that implicit messages are possible in pi calculi as well as concurrent lambda calculi. PLST (introduced in this thesis in section 2.4.8) is, like LAST, typed with binary session types, and therefore so is PIIM. Like IM, PIIM uses session type information to identify source locations where an implicit message is required to uphold duality between communicating processes. As with IM to LAST, PIIM derives its semantics by translation to PLST, in which implicit messages are converted to standard message exchanges. PIIM is proved type-safe via this translation.

4.1.1 Outline

Section 4.2 introduces PIIM with an example, section 4.3 introduces the syntax of PIIM, section 4.4 discusses typing and translation of PIIM to PLST, and section 4.5 covers our proof of the type safety of PIIM.

4.2 PIIM - An Example

Below is a PLST program akin to the example given in section 3.2.3, showing dependency injection for concurrent processes. The process *Manager* handles requests, e.g. http requests, on the channel *manager*, receiving first a private channel s over which communication with a single client can proceed. The client then delivers contextual information (bound to the variable ctx) over s to the manager, and based on that contextual information, the manager decides if it will handle the request in one of two ways - by delegating the handling of the request to either *Handler*1 or *Handler*2. It decides which handler to call based on some predicate $pred : CtxT \rightarrow \texttt{bool}$, whose details are unimportant. In either case, the manager creates a new channel ($h1$ or $h2$) to communicate with one of the

handlers, and sends the chosen handler a request, passing the new channel to the handler for private communication between the manager and the handler. The manager then passes the context to the handler, this being the dependency injection part of the protocol. After some computation abstracted by the action $doHandling1$ or $doHandling2$, the handler sends the result back to the manager on $h1$ or $h2$, which the manager forwards on to its client over s.

$$\begin{array}{rcl} Manager & = & !manager(s).s(ctx).\texttt{if } pred(ctx) \\ & & \texttt{then } (\nu h1)\overline{handler1}\langle h1\rangle.\overline{h1}\langle ctx\rangle.h1(result).\bar{s}\langle result\rangle \\ & & \texttt{else } (\nu h2)\overline{handler2}\langle h2\rangle.\overline{h2}\langle ctx\rangle.h2(result).\bar{s}\langle result\rangle \\ Handler1 & = & !handler1(h1).h1(ctx).doHandling1.\overline{h1}\langle result\rangle \\ Handler2 & = & !handler2(h2).h2(ctx).doHandling2.\overline{h2}\langle result\rangle \end{array}$$

The session type for the handler channels $h1$ and $h2$ are shown below, where $CtxT$ is the type of the passed contextual information, and $ResultT$ the type of the result computed by the handler:

$$h1, h2 : \texttt{lin?}CtxT.\texttt{lin!}ResultT.\texttt{end}$$

We can use implicit messages to handle the passing of context. The next example is a new version of the previous example, rewritten to use implicit messages to pass the context from the manager to the handlers. The handler's input operation for the context pass is annotated with the symbol $\wr$, indicating implicit input. The corresponding output in the manager is omitted.

$$\begin{array}{rcl} Manager & = & !manager(s).s(ctx).\texttt{if } pred(ctx) \\ & & \texttt{then } (\nu h1)\overline{handler1}\langle h1\rangle.h1(result).\bar{s}\langle result\rangle \\ & & \texttt{else } (\nu h2)\overline{handler2}\langle h2\rangle.h2(result).\bar{s}\langle result\rangle \\ Handler1 & = & !handler1(h1).h1(ctx)^{\wr}.doHandling1.\overline{h1}\langle result\rangle \\ Handler2 & = & !handler2(h2).h2(ctx)^{\wr}.doHandling2.\overline{h2}\langle result\rangle \end{array}$$

The new types of $h1$ and $h2$, shown below, are modified only in that their input of type $CtxT$ is now implicit.

$$h1, h2 : \texttt{lin?}^{\wr}CtxT.\texttt{lin!}ResultT.\texttt{end}$$

PIIM's type system is able to use this type information to rectify the apparent mismatch between the protocols on the manager's side and on the handler's side - the implicit receive operation communicates that a corresponding send operation must be inserted into the partner's protocol. When implicits are translated away, we are left with the original example.

4.3 The language PIIM

4.3.1 Syntax

As PIIM is an extension of PLST, it has very similar abstract syntax, differing only in the addition of implicit input $d(d)^{\wr}.P$. PIIM differs from PLST in the treatment of names - there are three *base* sets of names: *explicit names*, ranged over with x, x', ..., which are standard variable names used for non-implicit variables; *implicit names* ranged over with y, y', ..., which are variable names that replace implicit queries $\wr$, not appearing in source programs, but only in their translations (PLST programs); and finally the singleton set containing the implicit query $\wr$, which refers to all implicitly bound values. We then range over the union of the sets of implicit and explicit names, not including the implicit query, with z, z', ..., and we range over all names including implicit query with d, d'. We range over *processes* with P, Q, ... and *values* with v, v', ...

The grammar of PIIM terms is given in figure 4.1. In this and following figures in this chapter, we highlight PIIM's additions to PLST in red.

Our precise treatment of variable names allows us to express many different behaviours involving implicit variables with a minimal grammar. Using d, d', ... for the variable in restriction allows a fresh channel to be an implicitly bound value, and as the binder in implicit input, it allows a received channel to be bound implicitly. Naturally message contents are allowed to be implicitly bound values and so are also ranged over with d, d', ...

P, Q	$::=$	$P \mid Q$	*Parallel composition*
	$\mid$	$d(d).P$	*Input*
	$\mid$	$\overline{d}v.P$	*Output*
	$\mid$	$(\nu d)P$	*Restriction*
	$\mid$	`if` v `then` P `else` Q	*Conditional*
	$\mid$	$\mathbf{0}$	*Inaction*
	$\mid$	$d(d)^{\wr}.P$	*Implicit input*
v	$::=$	`true` $\mid$ `false`	*Boolean constants*
	$\mid$	x	*Names*
	$\mid$	$\wr$	*Implicit query*

FIGURE 4.1: Grammar of PIIM

4.3.2 Semantics

The semantics of PIIM, like IM to LAST, are given by first translating into PLST. Semantics of PLST are standard pi calculus semantics, which are given in figure 2.17, and extended

with standard rules for booleans: `if` b `then` P `else` Q reduces to P when b is `true`, and otherwise to Q.

4.4 Typing for PIIM

4.4.1 Types

As with PIIM's term syntax, we add very little to PLST's type syntax. We add only two new session types, $?^{\wr}T.S$ for implicit input and $!^{\wr}T.S$ for implicit output.

Figure 4.2 shows the grammar of types in PIIM. Note that as with IM, we consider only tail-recursive session types.

End point types:

S	::=	$q\ p$	*Qualified end point*
	|	`end`	*Used end point*
	|	a	*Type variable*
	|	$\mu a.S$	*Recursive end point*

Pre-end point types:

p	::=	$!T.S$	*Output*
	|	$?T.S$	*Input*
	|	$!^{\wr}T.S$	*Implicit output*
	|	$?^{\wr}T.S$	*Implicit input*

Types:

T	::=	`bool`	*Boolean*
	|	$(S, \overline{S})$	*Channel*

Qualifiers:

q	::=	`lin`	*Linear*
	|	`un`	*Unrestricted*

Contexts:

Γ	::=		*Empty context*
	|	$\Gamma, z : T$	*Variable binding*

FIGURE 4.2: Grammar of types in PIIM

We extend PLST's notion of a session type's dual, here written written $\overline{S}$, the type of a term that can safely interact with a term of type S. We define duality inductively on the syntax of types, extending PLST's definition to include implicit input and implicit output, which are each other's dual. The full definition is given in figure 4.3.

$$\overline{!S.\alpha} = ?S.\overline{\alpha} \qquad \overline{?S.\alpha} = !S.\overline{\alpha}$$

$$\overline{!^{\wr}S.\alpha} = ?^{\wr}S.\overline{\alpha} \qquad \overline{?^{\wr}S.\alpha} = !^{\wr}S.\overline{\alpha}$$

$$\overline{\mu t.\alpha} = \mu t.\overline{\alpha} \qquad \overline{t} = t \qquad \overline{\texttt{end}} = \texttt{end}$$

FIGURE 4.3: Type duality in PIIM

We define the *unrestricted* predicate `un` over types, end point types, and contexts. A type T or S is unrestricted, written $\texttt{un}(T)$ or $\texttt{un}(S)$, if the variable inhabiting it is used

more than once - that is to say that it is not linear. This concept comes from LLC and PLST, introduced in sections 2.3.7 and 2.4.8.

End point types are unrestricted if they are `end`, or if they are declared unlimited, i.e. $\mathtt{un}p$. Booleans are unlimited, as are channel types again if declared as such. The full definition of `un` for PIIM is given below:

Definition 6 (The `un` predicate)**.** For end point types S, $\mathtt{un}(S)$ holds in the cases:

$$\mathtt{un}(\mathtt{end}) \qquad \mathtt{un}(\mathtt{un}\ p) \qquad \mathtt{un}(\mu a.S) \text{ if } \mathtt{un}(S)$$

For types T, $\mathtt{un}(T)$ holds in the cases:

$$\mathtt{un}(\mathtt{bool}) \qquad \mathtt{un}((S, \overline{S})) \text{ if } \mathtt{un}(S), \mathtt{un}(\overline{S})$$

We extend the definition of `un` pointwise to environments Γ.

Again, PLST imports a concept from linear lambda calculus - *type splitting* (see section 2.3.7). Type splitting rules ensure that only unrestricted variables are copied arbitrarily in typing contexts. Copying linear variables is not allowed and thus it is ensured that the use of a single linear variable twice is not typable. The splitting rules for PIIM types, end points and contexts are given in figure 4.4.

The split of a session type S is $S \circ \mathtt{end}$ or $\mathtt{end} \circ S$, meaning that if a channel variable appears in two contexts, in one of those contexts it will have type `end` and therefore not be usable, but in the other will have type S and be usable according to its session type, thus protocol violations are prevented. Unrestricted session types are copied arbitrarily, as are booleans. Dual session type pairs are split inductively, as are contexts.

We define the addition $+$ on two contexts:

Definition 7 (The $+$ operation on contexts)**.** We define the partial operation $+$ as follows:

$$(\Gamma, z : T_1) + (z : T_2) = \Gamma, z : T_1 \circ T_2$$

4.4.2 Typing rules

Typing judgements in PIIM come in three forms, one for values v, one for processes P, and *binding judgements*, explained below. Judgements for values are of the form $\Gamma \vdash v : T \rightsquigarrow \widehat{v}$, meaning that under assumptions Γ, the value v has type T and is translated to the PLST term $\widehat{v}$. Judgements for processes are of the form $\Gamma \vdash P \rightsquigarrow \widehat{P}$, meaning that under assumptions Γ, the process P is well-typed and is translated to the PLST term $\widehat{P}$.

The typing rules for PIIM are given in figure 4.5. Rules [T-True], [T-False], [T-ExVar], [T-Inact], [T-Par], [T-Repl], [T-If], [T-Par] and [T-Res] are standard, all

End point type splitting rules

$$S = S \circ \texttt{end} \qquad S = \texttt{end} \circ S \qquad \texttt{un}\ p = \texttt{un}\ p \circ \texttt{un}\ p$$

Type splitting rules

$$\texttt{bool} = \texttt{bool} \circ \texttt{bool} \qquad \frac{R = R_1 \circ R_2 \quad S = S_1 \circ S_2}{(R, S) = (R_1, S_1) \circ (R_2, S_2)}$$

Context splitting rules

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad T = T_1 \circ T_2}{\Gamma, z : T = (\Gamma_1, z : T_1) \circ (\Gamma_2, z : T_2)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad T = T_2 \circ T_1}{\Gamma, z : T = (\Gamma_1, z : T_1) \circ (\Gamma_2, z : T_2)}$$

FIGURE 4.4: Type splitting rules

translations being simple homomorphisms. The rule [T-IMVAR] is equivalent to IM's rule [T-QUERY], checking the linearity constraints on Γ and choosing from Γ an implicit variable y to replace the implicit query $\wr$.

Binding judgements are two-place judgements of the form $d \rightsquigarrow z$. Every time a binder is encountered in the typing system, a binding rule will decide if that binder should be replaced with a fresh variable name (in the case that the binder is the implicit query $\wr$) or left unchanged. We allow use of $\wr$ as a binder in several places, for example in binding an input action $d(\wr).P$. PIIM has two binding rules - [T-IMBIND] generates a fresh name in such a case, and occurrences of the implicit query to be resolved to the variable there bound are replaced with the variable generated by the binding rule. The rule [T-EXBIND] leaves explicit variable bindings unchanged.

The final 8 rules whose names are given by the regular expression [T-(IM | EX)(IN | OUT)(L | R)] type communication actions for both implicit and explicit communication, and input and output, and come in left- and right-hand pairs. Rules for input ([T-EXINL], [T-EXINR], [T-IMINL], [T-IMINR]) split the conclusion context to type the communication channel and continuation process separately. The typing of the continuation adds the new binding, which in the implicit versions is accompanied a binding judgement to choose a new name for the binder should it be $\wr$. The output rules ([T-EXOUTL], [T-EXOUTR], [T-IMOUTL], [T-IMOUTR]) are similar, except that the conclusion context is split into three, to type the communication channel, continuation process *and* message contents. If the message content is a linear channel end point, the type splitting rules ensure that in the typing of the continuation process, the channel has type `end` and therefore cannot be communicated over, as it will have been delegated to the

communication partner.

4.4.3 Ambiguity

Like IM, PIIM implicit resolution has two sources of ambiguity: the choice of implicit variable, and the sequencing of multiple adjacent implicit communication actions, illustrated by the example in section 3.5.1. The same heuristics for resolving this ambiguity in IM are applicable to PIIM.

4.5 Type safety of PIIM

As with IM and LAST, we show type safety of PIIM by translation into PLST. We begin with a translation of PIIM types into PLST, written T^*, S^*. We then show that if a term is typable in PIIM, i.e. if $\Gamma \vdash P \rightsquigarrow \widehat{P}$, then the translation of the term is typable in PLST with types translated, which we write $\Gamma^* \vdash_{\bowtie} \widehat{P}$. We prove some intermediate lemmas to support this theorem. Note that where we refer to PLST typing rules, we use a subscript $\bowtie$, e.g. [T-INACT]$_{\bowtie}$. PIIM typing rules are not annotated, so [T-INACT] refers to PIIM's rule.

DEFINITION 8 (Translation of types)**.** We define the function $(\bullet)^*$, which translates PIIM types into PLST types by erasing occurrences of $\wr$.

$$
\begin{aligned}
\texttt{bool}^* &= \texttt{bool} \\
(S_1, S_2)^* &= (S_1^*, S_2^*) \\
\texttt{end}^* &= \texttt{end} \\
a^* &= a \\
(\mu a.S)^* &= \mu a.S^*
\end{aligned}
\qquad\qquad
\begin{aligned}
(q?T.S)^* &= q?T^*.S^* \\
(q!T.S)^* &= q!T^*.S^* \\
(q?^{\wr}T.S)^* &= q?T^*.S^* \\
(q!^{\wr}T.S)^* &= q!T^*.S^*
\end{aligned}
$$

We extend the definition of $(\bullet)^*$ pointwise to environments Γ.

LEMMA 12 (Preservation of un under translation)**.** If $\texttt{un}(T)$, then $\texttt{un}(T^*)$. Likewise, if $\texttt{un}(S)$, then $\texttt{un}(S^*)$. Finally, if $\texttt{un}(\Gamma)$, then $\texttt{un}(\Gamma^*)$.

Proof. Trivially by inductions on $\texttt{un}(T)$ and $\texttt{un}(S)$ applying definition 8. The result holds for $\texttt{un}(\Gamma)$ by pointwise extension. □

THEOREM 5 (Type-preserving translation of values)**.** Let Γ be a PIIM environment, v a PIIM value and T a PIIM type. If $\Gamma \vdash v : T \rightsquigarrow \widehat{v}$ then $\Gamma^* \vdash_{\bowtie} \widehat{v} : T^*$

Proof. By induction on typing derivations $\Gamma \vdash v : T \rightsquigarrow \widehat{v}$.

- Case $\Gamma \vdash \texttt{true} : \texttt{bool} \rightsquigarrow \texttt{true}$
  - We need to show $\Gamma^* \vdash_{\bowtie} \texttt{true} : \texttt{bool}^*$, or by definition 8, $\Gamma^* \vdash_{\bowtie} \texttt{true} : \texttt{bool}$
  - By inversion of [T-TRUE] we have $\texttt{un}(\Gamma)$, and then by lemma 12, $\texttt{un}(\Gamma^*)$
  - Finally by [T-TRUE]$_{\bowtie}$, the goal $\Gamma^* \vdash_{\bowtie} \texttt{true} : \texttt{bool}$ holds.
- Case $\Gamma \vdash \texttt{false} : \texttt{bool} \rightsquigarrow \texttt{false}$ – as above.
- Case $\Gamma, x : T \vdash x : T \rightsquigarrow x$
  - We need to show $(\Gamma, x : T)^* \vdash_{\bowtie} x : T^*$, or by definition 8, $\Gamma^*, x : T^* \vdash_{\bowtie} x : T^*$
  - By inversion of [T-EXVAR] we have $\texttt{un}(\Gamma)$, and then by lemma 12, $\texttt{un}(\Gamma^*)$
  - Finally by [T-VAR]$_{\bowtie}$, the goal $\Gamma^*, x : T^* \vdash_{\bowtie} x : T^*$ holds.
- Case $\Gamma, y : T \vdash \wr : T \rightsquigarrow y$
  - We need to show $(\Gamma, y : T)^* \vdash_{\bowtie} y : T^*$, or by definition 8, $\Gamma^*, y : T^* \vdash_{\bowtie} y : T^*$
  - By inversion of [T-IMVAR] we have $\texttt{un}(\Gamma)$, and then by lemma 12, $\texttt{un}(\Gamma^*)$
  - Finally by [T-VAR]$_{\bowtie}$, the goal $\Gamma^*, y : T^* \vdash_{\bowtie} y : T^*$ holds.

□

LEMMA 13 (Preservation of type splitting under translation)**.** For types T_1 and T_2, $(T_1 \circ T_2)^* = T_1^* \circ T_2^*$. Likewise for endpoints S_1 and S_2, $(S_1 \circ S_2)^* = S_1^* \circ S_2^*$. Finally for environments Γ_1 and Γ_2, $(\Gamma_1 \circ \Gamma_2)^* = \Gamma_1^* \circ \Gamma_2^*$.

Proof. By induction on derivations of the form $S = S_1 \circ S_2$, $T = T_1 \circ T_2$ and $\Gamma = \Gamma_1 \circ \Gamma_2$

- Case $S = S \circ \texttt{end}$
  - The rule $S = S \circ \texttt{end}$ implies $S^* = S^* \circ \texttt{end}$. By definition 8, $\texttt{end}^* = \texttt{end}$. Therefore $S^* = S^* \circ \texttt{end}^* = (S \circ \texttt{end})^*$.
- Case $S = \texttt{end} \circ S$
  - Similar to the above case.
- Case $\texttt{un}\ p = \texttt{un}\ p \circ \texttt{un}\ p$
  - By definition 8, $(\texttt{un}\ p)^* = \texttt{un}\ p^*$. Therefore the rule $\texttt{un}\ p = \texttt{un}\ p \circ \texttt{un}\ p$ implies $\texttt{un}\ p^* = \texttt{un}\ p^* \circ \texttt{un}\ p^* = (\texttt{un}\ p \circ \texttt{un}\ p)^*$
- Case $\texttt{bool} = \texttt{bool} \circ \texttt{bool}$
  - By definition 8, $\texttt{bool}^* = \texttt{bool}$. Therefore the rule $\texttt{bool} = \texttt{bool} \circ \texttt{bool}$ implies $\texttt{bool}^* = \texttt{bool}^* \circ \texttt{bool}^* = (\texttt{bool} \circ \texttt{bool})^*$

- Case $(R, S) = (R_1, S_1) \circ (R_2, S_2)$
  - By inversion, $R = R_1 \circ R_2$ and $S = S_1 \circ S_2$, and by induction, $R^* = R_1^* \circ R_2^*$ and $S^* = S_1^* \circ S_2^*$. The rule for channel type splitting implies $(R^*, S^*) = (R_1^*, S_1^*) \circ (R_2^*, S_2^*)$, and finally by definition 8, $(R, S)^* = (R_1, S_1)^* \circ (R_2, S_2)^* = ((R_1, S_1) \circ (R_2, S_2))^*$.
- Case $= \circ$
  - By definition 8, $^* = $. Therefore the rule $= \circ$ implies $^* = {}^* \circ {}^* = (\circ)^*$
- Case $\Gamma, x : T = (\Gamma_1, x : T_1) \circ (\Gamma_2, x : T_2)$
  - By inversion, $\Gamma = \Gamma_1 \circ \Gamma_2$, and either $T = T_1 \circ T_2$ or $T = T_2 \circ T_1$. By induction, $\Gamma^* = \Gamma_1^* \circ \Gamma_2^*$, and either $T^* = T_1^* \circ T_2^*$ or $T^* = T_2^* \circ T_1^*$. Then by either context splitting rule (depending on the split of T), $\Gamma^*, x : T^* = (\Gamma_1^*, x : T_1^*) \circ (\Gamma_2^*, x : T_2^*)$. Finally by definition 8, $(\Gamma, x : T)^* = (\Gamma_1, x : T_1)^* \circ (\Gamma_2, x : T_2)^* = ((\Gamma_1, x : T_1) \circ (\Gamma_2, x : T_2))^*$.

□

Lemma 14 (Preservation of context addition under translation)**.** If $\Gamma = \Gamma_1 + \Gamma_2$ then $\Gamma^* = \Gamma_1^* + \Gamma_2^*$.

Proof. Follows from definition 7 and lemma 13. □

Lemma 15 (Preservation of duality under translation)**.** If $S = \overline{S'}$ then $S^* = \overline{S'^*}$

Proof. A routine induction on $S^* = (\overline{S'})^*$ shows that reducing $(\bullet)^*$ yields dual types in all cases. □

Theorem 6 (Type-preserving translation of processes)**.** Let Γ be a PIIM environment and P a PIIM process. If $\Gamma \vdash P \rightsquigarrow \widehat{P}$ then $\Gamma^* \vdash_{\bowtie} \widehat{P}$

Proof. By induction on typing derivations $\Gamma \vdash P \rightsquigarrow \widehat{P}$.

- Case $\Gamma \vdash \mathbf{0} \rightsquigarrow \mathbf{0}$
  - We need to show $\Gamma^* \vdash_{\bowtie} \mathbf{0}$
  - By inversion of [T-Inact] we have $\mathtt{un}(\Gamma)$, and then by lemma 12, $\mathtt{un}(\Gamma^*)$
  - Finally by [T-Inact]$_{\bowtie}$, the goal $\Gamma^* \vdash_{\bowtie} \mathbf{0}$ holds.
- Case $\Gamma_1 \circ \Gamma_2 \vdash P \mid Q \rightsquigarrow \widehat{P} \mid \widehat{Q}$
  - We need to show $(\Gamma_1 \circ \Gamma_2)^* \vdash_{\bowtie} \widehat{P} \mid \widehat{Q}$, or by lemma 13, $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} \widehat{P} \mid \widehat{Q}$
  - By inversion of [T-Par], we have $\Gamma_1 \vdash P \rightsquigarrow \widehat{P}$ and $\Gamma_2 \vdash Q \rightsquigarrow \widehat{Q}$

  - By the induction hypothesis, $\Gamma_1^* \vdash_{\bowtie} \widehat{P}$ and $\Gamma_2^* \vdash_{\bowtie} \widehat{Q}$
  - Finally by [T-PAR]$_{\bowtie}$, the goal $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} \widehat{P} \mid \widehat{Q}$ holds.

- Case $\Gamma \vdash !P \rightsquigarrow !\widehat{P}$
  - We need to show $\Gamma^* \vdash_{\bowtie} !\widehat{P}$
  - By inversion of [T-REPL], we have:
    - $\Gamma \vdash P \rightsquigarrow \widehat{P}$, and then by the induction hypothesis, $\Gamma^* \vdash_{\bowtie} \widehat{P}$
    - $\mathtt{un}(\Gamma)$, and then by lemma 12, $\mathtt{un}(\Gamma^*)$
  - Finally by [T-REPL]$_{\bowtie}$, the goal $\Gamma^* \vdash_{\bowtie} !\widehat{P}$ holds.

- Case $\Gamma_1 \circ \Gamma_2 \vdash \mathtt{if}\ v\ \mathtt{then}\ P\ \mathtt{else}\ Q \rightsquigarrow \mathtt{if}\ \widehat{v}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{Q}$
  - We need to show $(\Gamma_1 \circ \Gamma_2)^* \vdash_{\bowtie} \mathtt{if}\ \widehat{e}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{Q}$, or by lemma 13, $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} \mathtt{if}\ \widehat{e}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{Q}$
  - By inversion of [T-IF], we have:
    - $\Gamma_1 \vdash v : \mathtt{bool} \rightsquigarrow \widehat{v}$, and by theorem 5, $\Gamma_1^* \vdash_{\bowtie} \widehat{v} : \mathtt{bool}^*$, and then by definition 8, $\Gamma_1^* \vdash_{\bowtie} \widehat{v} : \mathtt{bool}$
    - $\Gamma_2 \vdash P \rightsquigarrow \widehat{P}$, and by the induction hypothesis, $\Gamma_2^* \vdash_{\bowtie} \widehat{P}$
    - $\Gamma_2 \vdash Q \rightsquigarrow \widehat{Q}$, and by the induction hypothesis, $\Gamma_2^* \vdash_{\bowtie} \widehat{Q}$
  - Finally by [T-IF]$_{\bowtie}$, the goal $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} \mathtt{if}\ \widehat{e}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{Q}$ holds.

- Case $\Gamma \vdash (\nu d)P \rightsquigarrow (\nu z)\widehat{P}$
  - We need to show $\Gamma^* \vdash_{\bowtie} (\nu z)\widehat{P}$
  - By inversion of [T-RES], we have:
    - $d \rightsquigarrow z$
    - $\Gamma, z : (S, \overline{S}) \vdash P \rightsquigarrow \widehat{P}$, and by the induction hypothesis, $(\Gamma, z : (S, \overline{S}))^* \vdash_{\bowtie} \widehat{P}$
  - By definition 8 and lemma 15, $\Gamma^*, z : (S^*, \overline{S^*}) \vdash_{\bowtie} \widehat{P}$
  - Finally by [T-RES]$_{\bowtie}$, the goal $\Gamma^* \vdash_{\bowtie} (\nu z)\widehat{P}$ holds.

- Case $\Gamma_1 \circ \Gamma_2 \vdash d(d').P \rightsquigarrow z(z').\widehat{P}$
  - We need to show $(\Gamma_1 \circ \Gamma_2)^* \vdash_{\bowtie} z(z').\widehat{P}$, or by lemma 13, $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} z(z').\widehat{P}$
  - By inversion of [T-EXINL] (we could also apply inversion on [T-EXINR], and the resulting case is similar):
    - $\Gamma_1 \vdash d : (q?T.S, S') \rightsquigarrow z$
      - By theorem 5, $\Gamma_1^* \vdash_{\bowtie} z : (q?T.S, S')^*$

  - By definition 8, $\Gamma_1^* \vdash_{\bowtie} z : (q?T^*.S^*, S'^*)$
- $(\Gamma_2 + z : (S, S')), z' : T \vdash P \rightsquigarrow \widehat{P}$
  - By the induction hypothesis, $((\Gamma_2 + z : (S, S')), z' : T)^* \vdash_{\bowtie} \widehat{P}$
  - By definition 8 and lemma 14, $(\Gamma_2^* + z : (S^*, S'^*)), z' : T^* \vdash_{\bowtie} \widehat{P}$
- $d' \rightsquigarrow z'$
- $q = \mathtt{un} \Rightarrow q?T.S = S$

– Finally by [T-INL]$_{\bowtie}$, the goal $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} z(z').\widehat{P}$ holds.

- Case $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{d}v.P \rightsquigarrow \overline{z}\widehat{v}.\widehat{P}$

– We need to show $(\Gamma_1 \circ \Gamma_2 \circ \Gamma_3)^* \vdash_{\bowtie} \overline{z}\widehat{v}.\widehat{P}$, or by lemma 13, $\Gamma_1^* \circ \Gamma_2^* \circ \Gamma_3^* \vdash_{\bowtie} \overline{z}\widehat{v}.\widehat{P}$

– By inversion of [T-EXOUTL] (we could also apply inversion on [T-EXOUTR], and the resulting case is similar):

- $\Gamma_1 \vdash d : (q!T.S, S') \rightsquigarrow z$
  - By lemma 5, $\Gamma_1^* \vdash_{\bowtie} z : (q!T.S, S')^*$
  - By definition 8, $\Gamma_1^* \vdash_{\bowtie} z : (q!T^*.S^*, S'^*)$
- $\Gamma_2 \vdash v : T \rightsquigarrow \widehat{v}$
  - By lemma 5, $\Gamma_2^* \vdash_{\bowtie} \widehat{v} : T^*$
- $\Gamma_3 + d : (S, S') \vdash P \rightsquigarrow \widehat{P}$
  - By the induction hypothesis, $(\Gamma_3 + d : (S, S'))^* \vdash_{\bowtie} \widehat{P}$
  - By definition 8 and lemma 14, $\Gamma_3^* + d : (S^*, S'^*) \vdash_{\bowtie} \widehat{P}$
- $q = \mathtt{un} \Rightarrow q!T.S = S$

– Finally by [T-OUTL]$_{\bowtie}$, the goal $\Gamma_1^* \circ \Gamma_2^* \circ \Gamma_3^* \vdash_{\bowtie} \overline{z}\widehat{v}.\widehat{P}$ holds.

- Case $\Gamma_1 \circ \Gamma_2 \vdash d(d')^{\natural}.P \rightsquigarrow z(z').\widehat{P}$

– We need to show $(\Gamma_1 \circ \Gamma_2)^* \vdash_{\bowtie} z(z').\widehat{P}$, or by lemma 13, $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} z(z').\widehat{P}$

– By inversion of [T-IMINL] (we could also apply inversion on [T-IMINR], and the resulting case is similar):

- $\Gamma_1 \vdash d : (q?^{\natural}T.S, S') \rightsquigarrow z$
  - By theorem 5, $\Gamma_1^* \vdash_{\bowtie} z : (q?^{\natural}T.S, S')^*$
  - By definition 8, $\Gamma_1^* \vdash_{\bowtie} z : (q?T^*.S^*, S'^*)$
- $(\Gamma_2 + z : (S, S')), z' : T \vdash P \rightsquigarrow \widehat{P}$
  - By the induction hypothesis, $((\Gamma_2 + z : (S, S')), z' : T)^* \vdash_{\bowtie} \widehat{P}$
  - By definition 8 and lemma 14, $(\Gamma_2^* + z : (S^*, S'^*)), z' : T^* \vdash_{\bowtie} \widehat{P}$
- $d' \rightsquigarrow z'$
- $q = \mathtt{un} \Rightarrow q?^{\natural}T.S = S$

- By definition 8, $q?^{\wr}T.S = S \Rightarrow (q?^{\wr}T.S)^* = q?T^*.S^* = S^*$
- Therefore $q = \mathtt{un} \Rightarrow q?T^*.S^* = S^*$

– Finally by [T-INL]$_{\bowtie}$, the goal $\Gamma_1^* \circ \Gamma_2^* \vdash_{\bowtie} z(z').\widehat{P}$ holds.

- Case $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash P \rightsquigarrow \overline{z}y.\widehat{P}$

– We need to show $(\Gamma_1 \circ \Gamma_2 \circ \Gamma_3)^* \vdash_{\bowtie} \overline{z}y.\widehat{P}$, or by definition 8 and lemma 13, $\Gamma_1^* \circ \Gamma_2^* \circ \Gamma_3^* \vdash_{\bowtie} \overline{z}y.\widehat{P}$

– By inversion of [T-IMOUTL] (we could also apply inversion on [T-IMOUTR], and the resulting case is similar):

- $\Gamma_1 \vdash d : (q!^{\wr}T.S, S') \rightsquigarrow z$
  - By lemma 5, $\Gamma_1^* \vdash_{\bowtie} z : (q!^{\wr}T.S, S')^*$
  - By definition 8, $\Gamma_1^* \vdash_{\bowtie} z : (q!^{\wr}T^*.S^*, S'^*)$
- $\Gamma_2 \vdash \wr : T \rightsquigarrow y$
  - By lemma 5, $\Gamma_2^* \vdash_{\bowtie} y : T^*$
- $\Gamma_3 + z : (S, S') \vdash P \rightsquigarrow \widehat{P}$
  - By the induction hypothesis, $(\Gamma_3 + z : (S, S'))^* \vdash_{\bowtie} \widehat{P}$
  - By definition 8 and lemma 14, $\Gamma_3^* + z : (S^*, S'^*) \vdash_{\bowtie} \widehat{P}$
- $q = \mathtt{un} \Rightarrow q!^{\wr}T.S = S$
  - By definition 8, $q!^{\wr}T.S = S \Rightarrow (q!^{\wr}T.S)^* = q!T^*.S^* = S^*$
  - Therefore $q = \mathtt{un} \Rightarrow q!T^*.S^* = S^*$

– Finally by [T-OUTL]$_{\bowtie}$, the goal $\Gamma_1^* \circ \Gamma_2^* \circ \Gamma_3^* \vdash_{\bowtie} \overline{z}y.\widehat{P}$ holds.

□

4.6 Conclusion

PIIM offers an opportunity to focus on our novel programming language construct, implicit messages, in a reduced setting, without mixing in other constructs that already exist in other languages, such as implicit functions. PIIM is an ideal calculus for those wishing to study theoretical aspects of implicit messages, without being bothered by unrelated concepts such as lambda calculus - IM is a more realistic programming language, a basis for an implementation. The soundness result for PIIM, following the same approach as IM, shows the robustness of IM's translation based approach.

In the next chapter, we demonstrate that implicit messages are not limited to communication between two partners, but can be integrated into the multi-party conversations of multi-party session types.

$$\frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \texttt{true} : \texttt{bool} \rightsquigarrow \texttt{true}}\ [\text{T-TRUE}] \qquad \frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \texttt{false} : \texttt{bool} \rightsquigarrow \texttt{false}}\ [\text{T-FALSE}]$$

$$\frac{\mathrm{un}(\Gamma)}{\Gamma, x : T \vdash x : T \rightsquigarrow x}\ [\text{T-EXVAR}] \qquad \frac{\mathrm{un}(\Gamma)}{\Gamma, y : T \vdash \wr : T \rightsquigarrow y}\ [\text{T-IMVAR}]$$

$$x \rightsquigarrow x\ [\text{T-EXBIND}] \qquad \frac{y \text{ fresh}}{\wr \rightsquigarrow y}\ [\text{T-IMBIND}]$$

$$\frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathbf{0} \rightsquigarrow \mathbf{0}}\ [\text{T-INACT}] \qquad \frac{\Gamma_1 \vdash P \rightsquigarrow \widehat{P} \quad \Gamma_2 \vdash Q \rightsquigarrow \widehat{Q}}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q \rightsquigarrow \widehat{P} \mid \widehat{Q}}\ [\text{T-PAR}]$$

$$\frac{\Gamma \vdash P \rightsquigarrow \widehat{P} \quad \mathrm{un}(\Gamma)}{\Gamma \vdash !P \rightsquigarrow !\widehat{P}}\ [\text{T-REPL}]$$

$$\frac{\Gamma_1 \vdash v : \texttt{bool} \rightsquigarrow \widehat{v} \quad \Gamma_2 \vdash P \rightsquigarrow \widehat{P} \quad \Gamma_2 \vdash Q \rightsquigarrow \widehat{Q}}{\Gamma_1 \circ \Gamma_2 \vdash \texttt{if } v \texttt{ then } P \texttt{ else } Q \rightsquigarrow \texttt{if } \widehat{v} \texttt{ then } \widehat{P} \texttt{ else } \widehat{Q}}\ [\text{T-IF}]$$

$$\frac{\Gamma, z : (S, \overline{S}) \vdash P \rightsquigarrow \widehat{P} \quad d \rightsquigarrow z}{\Gamma \vdash (\nu d)P \rightsquigarrow (\nu z)\widehat{P}}\ [\text{T-RES}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (q?T.S, S') \rightsquigarrow z \quad d' \rightsquigarrow z' \\ (\Gamma_2 + z : (S, S')), z' : T \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q?T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash d(d').P \rightsquigarrow z(z').\widehat{P}}\ [\text{T-EXINL}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (S', q?T.S) \rightsquigarrow z \quad d' \rightsquigarrow z' \\ (\Gamma_2 + z : (S', S)), z' : T \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q?T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash d(d').P \rightsquigarrow z(z').\widehat{P}}\ [\text{T-EXINR}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (q!T.S, S') \rightsquigarrow z \\ \Gamma_2 \vdash v : T \rightsquigarrow \widehat{v} \quad \Gamma_3 + d : (S, S') \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q!T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{d}v.P \rightsquigarrow \overline{z}\widehat{v}.\widehat{P}}\ [\text{T-EXOUTL}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (S', q!T.S) \rightsquigarrow z \\ \Gamma_2 \vdash v : T \rightsquigarrow \widehat{v} \quad \Gamma_3 + d : (S', S) \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q!T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{d}v.P \rightsquigarrow \overline{z}\widehat{v}.\widehat{P}}\ [\text{T-EXOUTR}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (q?^{\wr}T.S, S') \rightsquigarrow z \quad d' \rightsquigarrow z' \\ (\Gamma_2 + z : (S, S')), z' : T \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q?^{\wr}T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash d(d')^{\wr}.P \rightsquigarrow z(z').\widehat{P}}\ [\text{T-IMINL}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (S', q?^{\wr}T.S) \rightsquigarrow z \quad d' \rightsquigarrow z' \\ (\Gamma_2 + z : (S', S)), z' : T \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q?^{\wr}T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash d(d')^{\wr}.P \rightsquigarrow z(z').\widehat{P}}\ [\text{T-IMINR}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (q!^{\wr}T.S, S') \rightsquigarrow z \\ \Gamma_2 \vdash \wr : T \rightsquigarrow y \quad \Gamma_3 + z : (S, S') \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q!^{\wr}T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash P \rightsquigarrow \overline{z}y.\widehat{P}}\ [\text{T-IMOUTL}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash d : (S', q!^{\wr}T.S) \rightsquigarrow z \\ \Gamma_2 \vdash \wr : T \rightsquigarrow y \quad \Gamma_3 + z : (S', S) \vdash P \rightsquigarrow \widehat{P} \quad q = \mathrm{un} \Rightarrow q!^{\wr}T.S = S\end{array}}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash P \rightsquigarrow \overline{z}y.\widehat{P}}\ [\text{T-IMOUTR}]$$

FIGURE 4.5: Typing system for PIIM

Chapter 5

Multiparty Asynchronous Sessions with Implicit Messages[1]

5.1 Introduction

In this chapter we introduce a multiparty session-typed pi calculus with implicit messages. We call the language *MultiParty Implicit Messages* (MPIM). As with IM and PIIM, we give meaning to MPIM programs by a type-directed translation to a base calculus. Our base calculus for MPIM is the multiparty session-typed pi calculus of [Coppo et al., 2015]. We term this base calculus MPST. We make one minor simplification to MPST in our usage of it as the base calculus here: we disallow multicast output. Disallowing multicast output allows us to make several inconsequential syntactic simplifications that aid in brevity. Since the resulting calculus with this simplification is a subset of the full calculus with multicast output, all the soundness results that we depend on still hold.

Multiparty session-typed pi calculus is a model of computation based purely on message passing, unlike LAST which is based on the lambda calculus. As such, the following formulation includes only implicit messages, and, like PIIM, omits implicit functions, which are not applicable in a language without lambda abstractions.

This formulation of implicit messages in a third setting further demonstrates the broad applicability of implicit messages to message-passing forms of computation, and further shows the robustness of the translation-based approach.

The example uses of implicit messages sketched in section 3.2 are applicable in MPIM as well as IM. MPIM allows for more flexibility in the use of implicit messages – for example, a process can interleave communication with a type class server and a third participant, which is not possible in IM (without sacrificing deadlock freedom).

[1] This chapter is adapted from and [Jeffery and Berger, 2019], a published paper co-authored with my supervisor, Dr. Martin BERGER. The sections of the paper that this chapter draws from are entirely original work.

5.2 MPIM - An Example

We now show an example MPIM protocol. In this simple protocol are four participants - participant 4 initiates the session and sends a message x of type α to participant 1. Participant 1 then *implicitly* passes x onto 2, who then implicitly passes it onto 3, who then passes it onto 4 explicitly. This protocol is captured by the global session type:

$$4 \to 1 : \langle \alpha \rangle.1 \to 2 :^{\wr} \langle \alpha \rangle.2 \to 3 :^{\wr} \langle \alpha \rangle.3 \to 4 : \langle \alpha \rangle.\mathtt{end}$$

Note the $\wr$ annotations on the implicit communication operations. An implementation of this protocol is shown below. Note that there are missing output operations in participants 1 and 2 – they are omitted since they are implicit. All communication occurs over the channel a bound with the same name in each participant process.

$$\begin{aligned}
&pingpong[1](a).a?(4, \wr).\mathbf{0} \mid \\
&pingpong[2](a).a?^{\wr}(1, \wr).\mathbf{0} \mid \\
&pingpong[3](a).a?^{\wr}(2, \wr).a!\langle 4, \wr \rangle.\mathbf{0} \mid \\
&\overline{pingpong}[4](a).a!\langle 2, x \rangle.a?(3, x).\mathbf{0}
\end{aligned}$$

The above implementation is translated to the following. The implicit outputs have been made explicit, and the implicit queries have been resolved to normal variables.

$$\begin{aligned}
&pingpong[1](a).a?(4, y).a!\langle 2, y \rangle.\mathbf{0} \mid \\
&pingpong[2](a).a?(1, y)a!\langle 3, y \rangle.\mathbf{0} \mid \\
&pingpong[3](a).a?(2, y).a!\langle 4, y \rangle.\mathbf{0} \mid \\
&\overline{pingpong}[4](a).a!\langle 2, x \rangle.a?(3, x).\mathbf{0}
\end{aligned}$$

The translated type for the above is given below – it matches the previous type except that the implicit annotations are removed.

$$4 \to 1 : \langle \alpha \rangle.1 \to 2 : \langle \alpha \rangle.2 \to 3 : \langle \alpha \rangle.3 \to 4 : \langle \alpha \rangle.\mathtt{end}$$

5.3 The language MPIM

5.3.1 Syntax

The grammar of MPIM is given in Figure 5.1. Note that x, y in P can also be $\wr$. We extend MPST with four new syntactic constructs. The first, implicit value reception, written $c?^{\wr}(\mathtt{p}, x).P$, can be read "on channel c, implicitly receive a value from participant $\mathtt{p}$, and bind it to the name x, then perform actions P". The second, implicit channel reception,

written $c?^{\wr}((\mathtt{q}, x)).P$ is similar, except that a channel is received as opposed to a value. The third, implicit channel hiding, written $(\nu\wr)P$, creates a fresh channel whose scope is P, accessible via an implicit query $\wr$. Finally, to the grammar of expressions, we add the implicit query $\wr$, whose behaviour is the same as in the languages IM and PIIM. As previously, we highlight MPIM's additions to MPST in red.

P	$::=$	$c?^{\wr}(\mathtt{p}, x).P$	Implicit Value Reception
	$\mid$	$c?^{\wr}((\mathtt{p}, x)).P$	Implicit Channel Reception
	$\mid$	$(\nu\wr)P$	Implicit Channel Hiding
	$\mid$	$c?(\mathtt{p}, x).P$	Value Reception
	$\mid$	$c?((\mathtt{q}, x)).P$	Channel Reception
	$\mid$	$(\nu a)P$	Channel Hiding
	$\mid$	$c!\langle \mathtt{p}, x\rangle.P$	Value Sending
	$\mid$	$c!\langle\langle \mathtt{p}, c'\rangle\rangle.P$	Channel Sending
	$\mid$	$c \oplus \langle \mathtt{p}, l\rangle.P$	Selection
	$\mid$	$c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I})$	Branching
	$\mid$	$P \mid Q$	Parallel composition
	$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional
	$\mid$	$\overline{u}[\mathtt{p}](y).P$	Multicast request
	$\mid$	$u[\mathtt{p}](y).P$	Accept
	$\mid$	def D in P	Recursion
	$\mid$	$X\langle e, c\rangle$	Process call
	$\mid$	$\mathbf{0}$	Inaction
D	$::=$	$X(x, y) = P$	Declaration
e	$::=$	$x \mid y$	Variable
	$\mid$	$\mathtt{true} \mid \mathtt{false}$	Boolean expression
	$\mid$	$e\ \mathtt{and}\ e'$	
	$\mid$	$\mathtt{not}\ e$	
x, y	$::=$	$\wr$	Implicit variable
	$\mid$	a	Explicit variable

FIGURE 5.1: Grammar of MPIM terms

5.3.2 Semantics

Unlike LAST, which has a single syntax definition, MPST has notions of *programmer syntax* and *runtime syntax*. The programmer syntax is intended to provide all constructs a programmer would need to write programs in MPST, and the runtime syntax, a superset of the programmer syntax, adds constructs like communication buffers needed only for program execution, that a programmer would not need, but are necessary to define

sensible asynchronous semantics. We define the runtime syntax (which is an extension of the syntax defined in figure 5.1 minus the additions of MPIM to MPST) in figure 5.2. [2]

As we derive semantics for IM by translation to LAST, we derive semantics for MPIM by translation to MPST, leveraging MPST's semantics, which are given in figure 5.3. Figure 5.4 defines structural congruence for MPST, on which depend the semantics. Note that the semantics are given with respect to the runtime syntax of MPST, and that we omit semantic rules for MPST that handle multicast output, since we do not allow multicast output in MPIM, and also give special cases of MPST's semantic rules for unicast output only, rather than the more general rules given in [Coppo et al., 2015].

P	$::=$	...	
	$\mid$	$(\nu s)P$	Session hiding
	$\mid$	$s : h$	Message queue
$\mathscr{E}$	$::=$	$[\bullet] \mid P \mid (\nu a : G)\mathscr{E}$	Evaluation context
	$\mid$	$(\nu s)\mathscr{E} \mid \text{def } D \text{ in } \mathscr{E}$	
	$\mid$	$\mathscr{E} \mid \mathscr{E}$	
c	$::=$	$y \mid s[\mathtt{p}]$	Channel
h	$::=$	$h \cdot m \mid \varnothing$	Message queue
m	$::=$	$(\mathtt{q}, \mathtt{p}, v)$	Value message
	$\mid$	$(\mathtt{q}, \mathtt{p}, s[\mathtt{p}'])$	Channel message
	$\mid$	$(\mathtt{q}, \mathtt{p}, l)$	Selection message

FIGURE 5.2: Runtime syntax for MPST

5.4 Types for MPIM

Figure 5.5 shows the grammar of types in MPIM. Note that as with IM and PIIM, we consider only tail-recursive session types.

We introduce two new session types. These are the dual types of implicit input and output, written $?^{\wr}(\mathtt{p}, U).T$ and $!^{\wr}\langle \mathtt{p}, U\rangle.T$ respectively, where the other participant's participant number is given by $\mathtt{p}$, the type of the exchanged value is U and the session type of the continuation is T. We also introduce one new global session type – implicit exchange $\mathtt{p} \to \mathtt{q} :^{\wr}\langle U\rangle.G$, representing implicit input and output at the global level, where the sender and receiver's participant numbers are given by $\mathtt{p}$ and $\mathtt{q}$ respectively, the type of the exchanged value is U and the global session type of the continuation is G.

[2] Note that it would have been possible to redefine LAST as having separate programmer and runtime syntax, and then define IM purely with respect to the programmer syntax of LAST, somewhat simplifying the definition of IM. We decided instead to match LAST as closely as possible for the sake of clarity, and so IM is defined with respect to the entire LAST language unlike MPIM and MPST.

$$a[1](y).P_1 \mid ... \mid a[n-1](y).P_{n-1} \mid \overline{a}[n](y).P_n \longrightarrow (\nu s)(P_1[s[1]/y] \mid ... \mid P_{n-1}[s[n-1]/y] \mid P_n[s[n]/y] \mid s : \varnothing) \quad [\textsc{Init}]$$

$$s[p]!\langle q, e\rangle.P \mid s : h \longrightarrow P \mid s : h \cdot (p, q, v) \text{ where } e \downarrow v \quad [\textsc{Send}]$$

$$s[p]!\langle\langle q, s'[p']\rangle\rangle.P \mid s : h \longrightarrow P \mid s : h \cdot (p, q, s'[p']) \quad [\textsc{Deleg}]$$

$$s[p] \oplus \langle q, l\rangle.P \mid s : h \longrightarrow P \mid s : h \cdot (p, q, l) \quad [\textsc{Sel}]$$

$$s[p]?(q, x).P \mid s : (q, p, v) \cdot h \longrightarrow P[v/x] \mid s : h \quad [\textsc{Rcv}]$$

$$s[p]?((q, y)).P \mid s : (q, p, s'[p']) \cdot h \longrightarrow P[s'[p']/y] \mid s : h \quad [\textsc{SRcv}]$$

$$s[p]\&(q, \{l_i : P_i\}_{i\in I}) \mid s : (q, p, l_j) \cdot h \longrightarrow P_j \mid s : h \text{ where } j \in I \quad [\textsc{Branch}]$$

$$\texttt{if } e \texttt{ then } P \texttt{ else } Q \longrightarrow P \text{ where } e \downarrow \texttt{true} \quad [\textsc{If-T}]$$

$$\texttt{if } e \texttt{ then } P \texttt{ else } Q \longrightarrow Q \text{ where } e \downarrow \texttt{false} \quad [\textsc{If-F}]$$

$$\text{def } X(x, y) = P \text{ in } (X\langle e, s[p]\rangle \mid Q) \longrightarrow \text{def } X(x, y) = P \text{ in } (P[v/x][s[p]/y] \mid Q) \text{ where } e \downarrow v \quad [\textsc{ProcCall}]$$

$$P \longrightarrow P' \Rightarrow \mathscr{E}[P] \longrightarrow \mathscr{E}[P'] \quad [\textsc{Ctxt}]$$

$$P \equiv P' \texttt{ and } P' \longrightarrow Q' \texttt{ and } Q \equiv Q' \Rightarrow P \longrightarrow Q \quad [\textsc{Str}]$$

FIGURE 5.3: Semantics of MPST terms

5.4.1 Duality

As with previous calculi, MPIM uses the notion of the dual of a session type S, here written $\overline{S}$, to mean session type that can safely interact with S. We define duality inductively on the syntax of types. The definition is given in Figure 5.6. We extend the definition of duality of MPST to include the new implicit exchange types.

5.4.2 Global Type Projection

Figure 5.7 shows the global projection of the generalised type G onto q, written $G \upharpoonright q$. We extend MPST's global projection to include our new global session type $p \to q :^{\wr}\langle U\rangle.G$

5.4.3 Partial Type Projection

Figure 5.8 shows the partial projection of the generalised type τ onto q, denoted by $\tau \upharpoonright q$. We extend MPST's partial projection to include our new session types $?^{\wr}(p, U).T$ and

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$P \mid (\nu r)Q \equiv (\nu r)(P \mid Q) \quad \text{if } r \notin \text{fn}(Q)$$

$$(\nu r)(\nu r')P \equiv (\nu r')(\nu r)P \qquad (\nu a : G)\mathbf{0} \equiv \mathbf{0} \qquad (\nu s)(s : \varnothing) \equiv \mathbf{0}$$
$$\text{where } r ::= a : G \mid s$$

$$\text{def } D \text{ in } \mathbf{0} \equiv \mathbf{0} \qquad \text{def } D \text{ in } (\nu r)P \equiv (\nu r)\text{def } D \text{ in } P \quad \text{if } r \notin \text{fn}(D)$$

$$(\text{def } D \text{ in } P) \mid Q \equiv \text{def } D \text{ in } (P \mid Q) \quad \text{if } \text{dpv}(D) \cap \text{dpv}(Q) = \varnothing$$

$$\text{def } D \text{ in } (\text{def } D' \text{ in } P) \equiv \text{def } D' \text{ in } (\text{def } D \text{ in } P)$$
$$\text{if } (\text{dpv}(D) \cup \text{fpv}(D)) \cap \text{dpv}(D') = \text{dpv}(D) \cap (\text{dpv}(D') \cup \text{fpv}(D')) = \varnothing$$

$$s : h \cdot (\mathtt{q}, \mathtt{p}, \zeta) \cdot (\mathtt{q}', \mathtt{p}', \zeta') \cdot h' \equiv s : h \cdot (\mathtt{q}', \mathtt{p}', \zeta') \cdot (\mathtt{q}, \mathtt{p}, \zeta) \cdot h' \quad \text{if } \mathtt{p} \neq \mathtt{p}' \text{ or } \mathtt{q} \neq \mathtt{q}'$$
$$\text{where } \zeta ::= v \mid s[\mathtt{p}] \mid l$$

$$P \equiv P' \Rightarrow \mathscr{E}[P] \equiv \mathscr{E}[P']$$

FIGURE 5.4: Structural congruence for MPST terms

$!^{\wr} p.U.T.$

5.5 Translation from MPIM to MPST

The typing and translation rules for MPIM expressions are given in Figure 5.9, and the rules for processes in Figure 5.10.

Our type system utilises two *binding rules* [IMBIND, EXBIND] which use a two place judgement of the form $x \rightsquigarrow a$. Since the binders in MPIM are allowed to be $\wr$, binding a received message to the implicit scope in the case of input, and binding a fresh channel to the implicit scope in the case of restriction, we must convert $\wr$ binders to standard names in the results of translation to MPST. Our binding rules handle the two possible cases: [EXBIND], when the binder is a normal name, leaves the binder unchanged. [IMBIND], which handles $\wr$, gives us a fresh standard name to replace $\wr$ in the translation. We add the fresh name to the typing environment, which allows us to replace $\wr$ with standard names in the components of the syntactic construct we are typing. Including binding premises in our typing rules for syntactic constructs with binders allows us to avoid duplicating typing rules. We avoid having separate rules for each construct with a binder, one for standard binders and one for implicit binders $\wr$.

$$
\begin{array}{rcll}
S & ::= & \texttt{bool} \mid \ldots \mid G & \text{Sorts} \\
U & ::= & S \mid \mathcal{T} & \text{Exchange types} \\
T & ::= & ?^{\wr}(\mathtt{p}, U).T & \text{Implicit Input} \\
 & \mid & !^{\wr}\langle \mathtt{p}, U\rangle.T & \text{Implicit Output} \\
 & \mid & ?(\mathtt{p}, U).T & \text{Explicit Input} \\
 & \mid & !\langle \mathtt{p}, U\rangle.T & \text{Explicit Output} \\
 & \mid & \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle & \text{Selection} \\
 & \mid & \&(\mathtt{p}, \{l_i : T_i\}_{i\in I}) & \text{Branching} \\
 & \mid & \mu t.T \mid t & \text{Recursion} \\
 & \mid & \texttt{end} & \text{Inaction} \\
G & ::= & \mathtt{p} \to \mathtt{q} :^{\wr}\langle U\rangle.G & \text{Implicit Exchange} \\
 & \mid & \mathtt{p} \to \mathtt{q} : \langle U\rangle.G & \text{Explicit Exchange} \\
 & \mid & \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i\in I} & \text{Branching} \\
 & \mid & \mu t.G \mid t & \text{Recursion} \\
 & \mid & \texttt{end} & \text{Inaction}
\end{array}
$$

FIGURE 5.5: Grammar of MPIM types

$$\texttt{end} \bowtie \texttt{end} \qquad t \bowtie t \qquad \mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow \mu t.\mathfrak{T} \bowtie \mu t.\mathfrak{T}'$$

$$\mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow !U.\mathfrak{T} \bowtie ?U.\mathfrak{T}' \qquad \mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow ?U.\mathfrak{T} \bowtie !U.\mathfrak{T}'$$

$$\mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow !^{\wr}U.\mathfrak{T} \bowtie ?^{\wr}U.\mathfrak{T}' \qquad \mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow ?^{\wr}U.\mathfrak{T} \bowtie !^{\wr}U.\mathfrak{T}'$$

$$\forall i \in I.\mathfrak{T}_i \bowtie \mathfrak{T}'_i \Longrightarrow \oplus\{l_i : \mathfrak{T}_i\}_{i\in I} \bowtie \&\{l_i : \mathfrak{T}'_i\}_{i\in I}$$

$$\exists i \in I.l = l_i \wedge \mathfrak{T} \bowtie \mathfrak{T}_i \Longrightarrow \oplus l;\mathfrak{T} \bowtie \&\{l_i : \mathfrak{T}_i\}_{i\in I}$$

FIGURE 5.6: Duality for MPIM session types

The rules [IMRCV] and [IMSRCV] type implicit value and channel input respectively. The premises $x \rightsquigarrow a$, $y \rightsquigarrow a$ replace implicit binders $\wr$ with fresh names where necessary.

The rule [IMNAME] functions similarly to IM's rule [T-QUERY], choosing a type-appropriate value to insert in place of an implicit query.

The rules [IMSEND] and [IMDELEG] synthesise outputs of values and channels respectively, where they are guided to do so by the appropriate process types. The premise $\Gamma \vdash \wr : S \rightsquigarrow y$ in [IMSEND] chooses an implicit value of the appropriate type to be send. [IMDELEG] has no such premise and instead chooses an unconsumed channel c from the session environment for delegation.

The rules [MCAST] and [MACC] type session request and acceptance respectively,

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

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

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

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

FIGURE 5.7: Global MPIM Type Projection

and are very similar to their counterpart rules in MPST, with the exception that the judgements are extended with translations to MPST terms.

The rule [NRES] handles channel restriction/creation. Again a binding premise replaces $\wr$ with standard names where necessary.

5.5.1 Translation of types

As with IM to LAST, we include a function to translate from MPIM types to MPST types. We use this function in §5.6 as part of our soundness theorem. Our translation for session types is similar to the translation for IM's session types, but we also extend the translation to global types.

DEFINITION 9 (Translation of types)**.** We define the *translation* of an MPIM type to a standard MPST type, written $\lceil S \rceil$, in figure 5.11. We extend the definition of $\lceil \bullet \rceil$ pointwise to standard environments Γ and session environments Δ.

5.6 Runtime safety of MPIM

In demonstrating the runtime safety of MPIM, our approach is similar to that used in demonstrating the runtime safety of IM: We show that if we can derive $\Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P}$, then $\widehat{P}$ can be typed suitably in MPST, according to the typing rules in [Coppo et al., 2015].

$$(!\langle \mathtt{p}, U\rangle.T) \upharpoonright \mathtt{q} = \begin{cases} !U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases} \qquad (?(\mathtt{p}, U).T) \upharpoonright \mathtt{q} = \begin{cases} ?U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases}$$

$$(\oplus\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle) \upharpoonright \mathtt{q} = \begin{cases} \oplus\{l_i : T_i \upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p} \\ T_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p} \text{ and } \forall i, j \in I.\ T_i \upharpoonright \mathtt{q} = T_j \upharpoonright \mathtt{q} \end{cases}$$

$$(\&(\mathtt{p}, \{l_i : T_i\}_{i\in I})) \upharpoonright \mathtt{q} = \begin{cases} \&\{l_i : T_i \upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p} \\ T_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p} \text{ and } \forall i, j \in I.\ T_i \upharpoonright \mathtt{q} = T_j \upharpoonright \mathtt{q} \end{cases}$$

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

$$(!^{\wr} p.U.T) \upharpoonright \mathtt{q} = \begin{cases} !^{\wr} U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases} \qquad (?^{\wr}(\mathtt{p}, U).T) \upharpoonright \mathtt{q} = \begin{cases} ?^{\wr} U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{p} = \mathtt{q} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases}$$

FIGURE 5.8: Partial MPIM Type Projection

$$\frac{}{x \rightsquigarrow x}\ [\textsc{ExBind}] \qquad \frac{x \text{ fresh}}{\wr \rightsquigarrow x}\ [\textsc{ImBind}]$$

$$\frac{}{\Gamma, y : S \vdash \wr : S \rightsquigarrow y}\ [\textsc{ImName}] \qquad \frac{}{\Gamma, x : S \vdash x : S \rightsquigarrow x}\ [\textsc{ExName}]$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool} \rightsquigarrow \mathtt{true}}\ [\textsc{True}] \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool} \rightsquigarrow \mathtt{false}}\ [\textsc{False}]$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \rightsquigarrow \widehat{e} \qquad \Gamma \vdash e' : \mathtt{bool} \rightsquigarrow \widehat{e}'}{\Gamma \vdash e \text{ and } e' : \mathtt{bool} \rightsquigarrow \widehat{e} \text{ and } \widehat{e}'}\ [\textsc{And}]$$

FIGURE 5.9: Typing and translation rules for MPIM expressions

Again we make this precise using a translation function $\lceil \cdot \rceil$, which translates MPIM's types to standard MPST types, defined in section 5.5.1.

THEOREM 7 (Type-preserving translation of expressions)**.** If $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$ then $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \lceil S \rceil$.

Proof. By induction on typing judgements for expressions $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$. We omit judgements for syntax present in standard MPST as these cases are homomorphic.

- Case $\Gamma, y : S \vdash \wr : S \rightsquigarrow y$
  - To show: $\lceil \Gamma, y : S \rceil \vdash_{\text{MPST}} y : \lceil S \rceil$

  - Or by Definition 9: $\lceil\Gamma\rceil, y : \lceil S\rceil \vdash_{\text{MPST}} y : \lceil S\rceil$
- The goal follows immediately from $[\text{NAME}]_{\text{MPST}}$.

- Case $\Gamma, x : S \vdash x : S \rightsquigarrow x$
  - To show: $\lceil\Gamma, x : S\rceil \vdash_{\text{MPST}} x : \lceil S\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil, x : \lceil S\rceil \vdash_{\text{MPST}} x : \lceil S\rceil$
  - The goal follows immediately from $[\text{NAME}]_{\text{MPST}}$.

- Case $\Gamma \vdash \texttt{true} : \texttt{bool} \rightsquigarrow \texttt{true}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \texttt{true} : \lceil\texttt{bool}\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \texttt{true} : \texttt{bool}$
  - The goal follows immediately from $[\text{BOOL}]_{\text{MPST}}$.

- Case $\Gamma \vdash \texttt{false} : \texttt{bool} \rightsquigarrow \texttt{false}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \texttt{false} : \lceil\texttt{bool}\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \texttt{false} : \texttt{bool}$
  - The goal follows immediately from $[\text{BOOL}]_{\text{MPST}}$.

- Case $\Gamma \vdash e \texttt{ and } e' : \texttt{bool} \rightsquigarrow \widehat{e} \texttt{ and } \widehat{e'}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e} \texttt{ and } \widehat{e'} : \lceil\texttt{bool}\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e} \texttt{ and } \widehat{e'} : \texttt{bool}$
  - By inversion of [AND]:
    - $\Gamma \vdash e : \texttt{bool} \rightsquigarrow \widehat{e}$
      - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e} : \lceil\texttt{bool}\rceil$
      - By Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e} : \texttt{bool}$
    - $\Gamma \vdash e' : \texttt{bool} \rightsquigarrow \widehat{e'}$
      - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e'} : \lceil\texttt{bool}\rceil$
      - By Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e'} : \texttt{bool}$
  - The goal then follows from $[\text{AND}]_{\text{MPST}}$.

□

LEMMA 16 (Preservation of participant numbers)**.** If $\texttt{p} = mp(G)$ then $\texttt{p} = mp(\lceil G\rceil)$. If $\texttt{p} < mp(G)$ then $\texttt{p} < mp(\lceil G\rceil)$.

Proof. Directly from the definition of $\lceil G\rceil$. □

THEOREM 8 (Type-preserving translation of pure processes)**.** If $\Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P}$ then $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil$.

Proof. By induction on typing judgements for pure processes $\Gamma \vdash P : \Delta \rightsquigarrow \widehat{P}$. We omit judgements for syntax present in standard MPST as these cases are homomorphic.

- Case $\Gamma \vdash c?^{\wr}(\mathfrak{q}, x).P \triangleright \Delta, c :?^{\wr}(\mathfrak{q}, S).T \rightsquigarrow c?(\mathfrak{q}, a).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?(\mathfrak{q}, x).\widehat{P} \triangleright \lceil\Delta, c :?^{\wr}(\mathfrak{q}, S).T\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?(\mathfrak{q}, x).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathfrak{q}, \lceil S\rceil).\lceil T\rceil$
  - By inversion of [IMRCV]:
    - $\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
      - And by the induction hypothesis: $\lceil\Gamma, a : S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
      - By Definition 9: $\lceil\Gamma\rceil, a : \lceil S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
      - By $[\text{RCV}]_{\text{MPST}}$: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?(\mathfrak{q}, a).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathfrak{q}, \lceil S\rceil).\lceil T\rceil$
    - $x \rightsquigarrow a$, and then $a \neq \wr$ and either:
      - $a = x$, and the goal holds by $[\text{RCV}]_{\text{MPST}}$ with $c?(\mathfrak{q}, a).\widehat{P} = c?(\mathfrak{q}, x).\widehat{P}$
      - a fresh, and the goal holds by $[\text{RCV}]_{\text{MPST}}$ with $c?(\mathfrak{q}, a).\widehat{P} \equiv_{\alpha} c?(\mathfrak{q}, x).\widehat{P}$
- Case $\Gamma \vdash c?^{\wr}((\mathfrak{q}, y)).P \triangleright \Delta, c :?^{\wr}(\mathfrak{q}, \mathcal{T}).T \rightsquigarrow c?((\mathfrak{q}, a)).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?((\mathfrak{q}, a)).\widehat{P} \triangleright \lceil\Delta, c :?^{\wr}(\mathfrak{q}, \mathcal{T}).T\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?((\mathfrak{q}, a)).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathfrak{q}, \lceil\mathcal{T}\rceil).\lceil T\rceil$
  - By inversion of [IMSRCV]:
    - $\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T, y : \mathcal{T}\rceil$
      - By Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil, y : \lceil\mathcal{T}\rceil$
      - By $[\text{SRCV}]_{\text{MPST}}$: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?((\mathfrak{q}, y)).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathfrak{q}, \lceil\mathcal{T}\rceil).\lceil T\rceil$
    - $y \rightsquigarrow a$, and then $a \neq \wr$ and either...
      - $a = y$, and the goal holds by $[\text{SRCV}]_{\text{MPST}}$ with $c?((\mathfrak{q}, a)).\widehat{P} = c?((\mathfrak{q}, y)).\widehat{P}$
      - a fresh, and the goal holds by $[\text{SRCV}]_{\text{MPST}}$ with $c?((\mathfrak{q}, a)).\widehat{P} \equiv_{\alpha} c?((\mathfrak{q}, y)).\widehat{P}$
- Case $\Gamma \vdash P \triangleright \Delta, c :!^{\wr}\langle \mathfrak{p}, S\rangle.T \rightsquigarrow c!\langle \mathfrak{p}, y\rangle.\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathfrak{p}, y\rangle.\widehat{P} \triangleright \lceil\Delta, c :!^{\wr}\langle \mathfrak{p}, S\rangle.T\rceil$
  - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathfrak{p}, y\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!^{\wr}\langle \mathfrak{p}, \lceil S\rceil\rangle.\lceil T\rceil$
  - By inversion of [IMSEND]:
    - $\Gamma \vdash \wr : S \rightsquigarrow y$
      - By Theorem 7: $\lceil\Gamma\rceil \vdash_{\text{MPST}} y : \lceil S\rceil$

  - $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
    - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
    - By Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $[\text{SEND}]_{\text{MPST}}$.

- Case $\Gamma \vdash P \triangleright \Delta, c :!^{\wr}\langle \mathtt{p}, \mathcal{T}\rangle.T, i : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, i\rangle\rangle.P$
  - To show $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, i\rangle\rangle.\widehat{P} \triangleright \lceil\Delta, c :!^{\wr}\langle \mathtt{p}, \mathcal{T}\rangle.T, i : \mathcal{T}\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, i\rangle\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!\langle \mathtt{p}, \lceil\mathcal{T}\rceil\rangle.\lceil T\rceil, i : \lceil\mathcal{T}\rceil$
  - By inversion of [IMDELEG]: $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
    - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
    - By Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $[\text{DELEG}]_{\text{MPST}}$.

- Case $\Gamma \vdash c?(\mathtt{q}, x).P \triangleright \Delta, c :?(\mathtt{q}, S).T \rightsquigarrow c?(\mathtt{q}, a).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?(\mathtt{q}, a).\widehat{P} \triangleright \lceil\Delta, c :?(\mathtt{q}, S).T\rceil$
    - Or by Definition 9:$\lceil\Gamma\rceil \vdash_{\text{MPST}} c?(\mathtt{q}, a).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathtt{q}, \lceil S\rceil).\lceil T\rceil$
  - By inversion of [RCV]:
    - $\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
      - And by the induction hypothesis: $\lceil\Gamma, a : S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
      - And by Definition 9: $\lceil\Gamma\rceil, a : \lceil S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
    - $x \rightsquigarrow a$, and then $a \neq \wr$ and either...
      - $a = x$, and the goal holds by $[\text{RCV}]_{\text{MPST}}$ with $c?(\mathtt{q}, a).\widehat{P} = c?(\mathtt{q}, x).\widehat{P}$
      - a fresh, and the goal holds by $[\text{RCV}]_{\text{MPST}}$ with $c?(\mathtt{q}, a).\widehat{P} \equiv_{\alpha} c?(\mathtt{q}, x).\widehat{P}$

- Case $\Gamma \vdash c?((\mathtt{q}, y)).P \triangleright \Delta, c :?(\mathtt{q}, \mathcal{T}).T \rightsquigarrow c?((\mathtt{q}, a)).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?((\mathtt{q}, a)).\widehat{P} \triangleright \lceil\Delta, c :?(\mathtt{q}, \mathcal{T}).T\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?((\mathtt{q}, a)).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathtt{q}, \lceil\mathcal{T}\rceil).\lceil T\rceil$
  - By inversion of [SRCV]:
    - $\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P}$
      - Then by the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T, y : \mathcal{T}\rceil$
      - Then by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil, y : \lceil\mathcal{T}\rceil$
    - $y \rightsquigarrow a$, and then $a \neq \wr$ and either...
      - $a = y$, and the goal holds by $[\text{SRCV}]_{\text{MPST}}$ with $c?((\mathtt{q}, a)).\widehat{P} = c?((\mathtt{q}, y)).\widehat{P}$
      - a fresh, and the goal holds by $[\text{SRCV}]_{\text{MPST}}$ with $c?((\mathtt{q}, a)).\widehat{P} \equiv_{\alpha} c?((\mathtt{q}, y)).\widehat{P}$

- Case $\Gamma \vdash c!\langle \mathtt{p}, e\rangle.P \triangleright \Delta, c :!\langle \mathtt{p}, S\rangle.T \rightsquigarrow c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P} \triangleright \lceil\Delta, c :!\langle \mathtt{p}, S\rangle.T\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!\langle \mathtt{p}, \lceil S\rceil\rangle.\lceil T\rceil$
  - By inversion of [SEND]:
    - $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$
      - Then by Theorem 7: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e} : \lceil S\rceil$
    - $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
      - Then by the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
      - Then by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $[\text{SEND}]_{\text{MPST}}$.
- Case $\Gamma \vdash c!\langle\langle \mathtt{p}, c'\rangle\rangle.P \triangleright \Delta, c :!\langle \mathtt{p}, \mathcal{T}\rangle.T, c' : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P} \triangleright \lceil\Delta, c :!\langle \mathtt{p}, \mathcal{T}\rangle.T, c' : \mathcal{T}\rceil$
    - Or by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!\langle \mathtt{p}, \lceil\mathcal{T}\rceil\rangle.\lceil T\rceil, c' : \lceil\mathcal{T}\rceil$
  - By inversion on [DELEG]: $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
    - Then by the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
    - Then by Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $[\text{DELEG}]_{\text{MPST}}$.
- Case $\Gamma \vdash \overline{u}[\mathtt{p}](y).P \triangleright \Delta \rightsquigarrow \overline{u}[\mathtt{p}](y).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \overline{u}[\mathtt{p}](y).\widehat{P} \triangleright \lceil\Delta\rceil$
  - By inversion of [MCAST]:
    - $\Gamma \vdash u : G \rightsquigarrow u$
      - By Theorem 7: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u : \lceil G\rceil$
      - By Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u : \lceil G\rceil$
    - $\Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p} \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, y : G \upharpoonright \mathtt{p}\rceil$
      - By Definition 9: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, y : \lceil G \upharpoonright \mathtt{p}\rceil$
    - $\mathtt{p} = mp(G)$
      - by Lemma 16: $\mathtt{p} = mp(\lceil G\rceil)$
  - The goal then holds by $[\text{MCAST}]_{\text{MPST}}$.
- Case $\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta \rightsquigarrow u[\mathtt{p}](y).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u[\mathtt{p}](y).\widehat{P} \triangleright \lceil\Delta\rceil$

  - By inversion of [MACC]:
    - $\Gamma \vdash u : G \rightsquigarrow u$
      - By Theorem 7: $\lceil \Gamma \rceil \vdash_{\text{MPST}} u : \lceil G \rceil$
      - By Definition 9: $\lceil \Gamma \rceil \vdash_{\text{MPST}} u : \lceil G \rceil$
    - $\Gamma \vdash P \rhd \Delta, y : G \upharpoonright \mathtt{p} \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil \Delta, y : G \upharpoonright \mathtt{p} \rceil$
      - By Definition 9: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil \Delta \rceil, y : \lceil G \upharpoonright \mathtt{p} \rceil$
    - $\mathtt{p} < mp(G)$
      - By Lemma 16: $\mathtt{p} < mp(\lceil G \rceil)$
  - The goal then holds by $[\text{MACC}]_{\text{MPST}}$.

- Case $\Gamma \vdash (\nu x)P \rhd \Delta \rightsquigarrow (\nu a)\widehat{P}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} (\nu a)\widehat{P} \rhd \lceil \Delta \rceil$
  - By inversion of [NRES]:
    - $\Gamma, a : G \vdash P \rhd \Delta \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil \Gamma, a : G \rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil \Delta \rceil$
      - By Definition 9: $\lceil \Gamma \rceil, a : \lceil G \rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil \Delta \rceil$
    - $x \rightsquigarrow a$, and then $a \neq \wr$ and either...
      - $a = x$, and the goal holds by $[\text{NRES}]_{\text{MPST}}$ with $(\nu a)\widehat{P} = (\nu x)\widehat{P}$
      - a fresh, and the goal holds by $[\text{NRES}]_{\text{MPST}}$ with $(\nu a)\widehat{P} \equiv_\alpha (\nu x)\widehat{P}$

- Case $\Gamma \vdash c \oplus \langle \mathtt{p}, l_j \rangle.P \rhd \Delta, c : \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I} \rangle \rightsquigarrow c \oplus \langle \mathtt{p}, l_j \rangle.\widehat{P}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c \oplus \langle \mathtt{p}, l_j \rangle.\widehat{P} \rhd \lceil \Delta, c : \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I} \rangle \rceil$
    - Or by Definition 9: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c \oplus \langle \mathtt{p}, l_j \rangle.\widehat{P} \rhd \lceil \Delta \rceil, c : \oplus\langle \mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I} \rangle$
  - By inversion of [SEL]:
    - $j \in I$
    - $\Gamma \vdash P \rhd \Delta, c : T_j \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil \Delta, c : T_j \rceil$
      - By Definition 9: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil \Delta \rceil, c : \lceil T_j \rceil$
  - The goal then holds by $[\text{SEL}]_{\text{MPST}}$.

- Case $\Gamma \vdash c\&(\mathtt{p}, \{l_i : P_i\}_{i \in I}) \rhd \Delta, c : \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) \rightsquigarrow c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i \in I})$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i \in I}) \rhd \lceil \Delta, c : \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) \rceil$
    - Or by Definition 9: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i \in I}) \rhd \lceil \Delta \rceil, c : \&(\mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I})$
  - By inversion of [BRANCH]: $\forall i \in I. \quad \Gamma \vdash P_i \rhd \Delta, c : T_i \rightsquigarrow \widehat{P}_i$

  - By the induction hypothesis: $\forall i \in I. \quad \lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P}_i \triangleright \lceil \Delta, c : T_i \rceil$
  - By Definition 9: $\forall i \in I. \quad \lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P}_i \triangleright \lceil \Delta \rceil, c : \lceil T_i \rceil$
- The goal then follows from $[\text{BRANCH}]_{\text{MPST}}$.

- Case $\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P' \triangleright \Delta \rightsquigarrow \texttt{if}\ \widehat{e}\ \texttt{then}\ \widehat{P}\ \texttt{else}\ \widehat{P}'$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \texttt{if}\ \widehat{e}\ \texttt{then}\ \widehat{P}\ \texttt{else}\ \widehat{P}' \triangleright \lceil \Delta \rceil$
  - By inversion of [IF]:
    - $\Gamma \vdash e : \texttt{bool} \rightsquigarrow \widehat{e}$
      - By Theorem 7: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \lceil \texttt{bool} \rceil$
      - By Definition 9: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \texttt{bool}$
    - $\Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta \rceil$
    - $\Gamma \vdash P' \triangleright \Delta \rightsquigarrow \widehat{P}'$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P}' \triangleright \lceil \Delta \rceil$
  - The goal then holds by $[\text{IF}]_{\text{MPST}}$.

- Case $\Gamma \vdash \mathbf{0} \triangleright \Delta \rightsquigarrow \mathbf{0}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathbf{0} \triangleright \lceil \Delta \rceil$
  - By inversion of [INACT]: Δ end only
    - By Definition 9: $\lceil \Delta \rceil$ end only
  - The goal then holds by $[\text{INACT}]_{\text{MPST}}$.

- Case $\Gamma, X : S\ T \vdash X\langle e, c \rangle \triangleright \Delta, c : T \rightsquigarrow X\langle \widehat{e}, c \rangle$
  - To show: $\lceil \Gamma, X : S\ T \rceil \vdash_{\text{MPST}} X\langle \widehat{e}, c \rangle \triangleright \lceil \Delta, c : T \rceil$
    - Or by Definition 9: $\lceil \Gamma \rceil, X : \lceil S \rceil\ \lceil T \rceil \vdash_{\text{MPST}} X\langle \widehat{e}, c \rangle \triangleright \lceil \Delta \rceil, c : \lceil T \rceil$
  - By inversion of [VAR]:
    - $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$
      - By Theorem 7, $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \lceil S \rceil$
    - Δ end only
      - By Definition 9, $\lceil \Delta \rceil$ end only
  - The goal then follows from $[\text{VAR}]_{\text{MPST}}$.

- Case $\Gamma \vdash \mathbf{def}\ X(x, y) = P\ \mathbf{in}\ Q \triangleright \Delta \rightsquigarrow \mathbf{def}\ X(a, b) = \widehat{P}\ \mathbf{in}\ \widehat{Q}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathbf{def}\ X(a, b) = \widehat{P}\ \mathbf{in}\ \widehat{Q} \triangleright \lceil \Delta \rceil$
  - By inversion of [DEF]:

- $\Gamma, X : S\ t, a : S \vdash P \triangleright b : T \rightsquigarrow \widehat{P}$
  - And by the induction hypothesis: $\lceil \Gamma, X : S\ t, a : S \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil b : T \rceil$
  - And by Definition 9: $\lceil \Gamma \rceil, X : \lceil S \rceil\ t, a : \lceil S \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright b : \lceil T \rceil$
- $\Gamma, X : S\ \mu t.T \vdash Q \triangleright \Delta \rightsquigarrow \widehat{Q}$
  - And by the induction hypothesis: $\lceil \Gamma, X : S\ \mu t.T \rceil \vdash_{\text{MPST}} \widehat{Q} \triangleright \lceil \Delta \rceil$
  - And by Definition 9: $\lceil \Gamma \rceil, X : \lceil S \rceil\ \mu t.\lceil T \rceil \vdash_{\text{MPST}} \widehat{Q} \triangleright \lceil \Delta \rceil$
- $x \rightsquigarrow a$, and then $a \neq \wr$ and either $a = x$ or a fresh
- $y \rightsquigarrow b$, and then $b \neq \wr$ and either $b = x$ or b fresh

– The goal then holds by $[\text{DEF}]_{\text{MPST}}$ with **def** $X(a, b) = \widehat{P}$ **in** $\widehat{Q}$ being either equal to, or α-equivalent to **def** $X(x, y) = \widehat{P}$ **in** $\widehat{Q}$.

□

5.7 Conclusion

In this chapter we have seen that implicit messages generalise from binary session-typed communication to multi-party session-typed communication, in a sound manner. This development allows for the incorporation of implicit messages into a much wider variety of real-world concurrent systems.

In the next chapter we refocus our attention to implicit functions. We show that they can be safely integrated into the object calculus DOT, the theoretical foundation for Scala, allowing DOT to model some of Scala's typical use cases for implicits.

$$\frac{\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P} \qquad x \rightsquigarrow a}{\Gamma \vdash c?^{\wr}(\mathtt{q}, x).P \triangleright \Delta, c : ?^{\wr}(\mathtt{q}, S).T \rightsquigarrow c?(\mathtt{q}, a).\widehat{P}} \quad [\textsc{ImRcv}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P} \qquad y \rightsquigarrow a}{\Gamma \vdash c?^{\wr}((\mathtt{q}, y)).P \triangleright \Delta, c : ?^{\wr}(\mathtt{q}, \mathcal{T}).T \rightsquigarrow c?((\mathtt{q}, a)).\widehat{P}} \quad [\textsc{ImSrcv}]$$

$$\frac{\Gamma \vdash \wr : S \rightsquigarrow y \qquad \Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash P \triangleright \Delta, c : !^{\wr}\langle \mathtt{p}, S\rangle.T \rightsquigarrow c!\langle \mathtt{p}, y\rangle.\widehat{P}} \quad [\textsc{ImSend}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash P \triangleright \Delta, c : !^{\wr}\langle \mathtt{p}, \mathcal{T}\rangle.T, i : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, i\rangle\rangle.\widehat{P}} \quad [\textsc{ImDeleg}]$$

$$\frac{\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P} \qquad x \rightsquigarrow a}{\Gamma \vdash c?(\mathtt{q}, x).P \triangleright \Delta, c :?(\mathtt{q}, S).T \rightsquigarrow c?(\mathtt{q}, a).\widehat{P}} \quad [\textsc{Rcv}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P} \qquad y \rightsquigarrow a}{\Gamma \vdash c?((\mathtt{q}, y)).P \triangleright \Delta, c :?(\mathtt{q}, \mathcal{T}).T \rightsquigarrow c?((\mathtt{q}, a)).\widehat{P}} \quad [\textsc{Srcv}]$$

$$\frac{\Gamma \vdash e : S \rightsquigarrow \widehat{e} \qquad \Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash c!\langle \mathtt{p}, e\rangle.P \triangleright \Delta, c :!\langle \mathtt{p}, S\rangle.T \rightsquigarrow c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P}} \quad [\textsc{Send}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash c!\langle\langle \mathtt{p}, c'\rangle\rangle.P \triangleright \Delta, c :!\langle \mathtt{p}, \mathcal{T}\rangle.T, c' : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P}} \quad [\textsc{Deleg}]$$

$$\frac{\Gamma \vdash u : G \rightsquigarrow u \qquad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p} \rightsquigarrow \widehat{P} \qquad \mathtt{p} = mp(G)}{\Gamma \vdash \overline{u}[\mathtt{p}](y).P \triangleright \Delta \rightsquigarrow \overline{u}[\mathtt{p}](y).\widehat{P}} \quad [\textsc{MCast}]$$

$$\frac{\Gamma \vdash u : G \rightsquigarrow u \qquad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p} \rightsquigarrow \widehat{P} \qquad \mathtt{p} < mp(G)}{\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta \rightsquigarrow u[\mathtt{p}](y).\widehat{P}} \quad [\textsc{MAcc}]$$

$$\frac{\Gamma, a : G \vdash P \triangleright \Delta \rightsquigarrow \widehat{P} \qquad x \rightsquigarrow a}{\Gamma \vdash (\nu x)P \triangleright \Delta \rightsquigarrow (\nu a)\widehat{P}} \quad [\textsc{NRes}] \qquad \frac{\Delta \text{ end only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta \rightsquigarrow \mathbf{0}} \quad [\textsc{Inact}]$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \rightsquigarrow \widehat{e} \qquad \Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P} \qquad \Gamma \vdash P' \triangleright \Delta \rightsquigarrow \widehat{P}'}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \triangleright \Delta \rightsquigarrow \mathtt{if}\ \widehat{e}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{P}'} \quad [\textsc{If}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T_j \rightsquigarrow \widehat{P} \qquad j \in I}{\Gamma \vdash c \oplus \langle \mathtt{p}, l_j\rangle.P \triangleright \Delta, c : \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle \rightsquigarrow c \oplus \langle \mathtt{p}, l_j\rangle.\widehat{P}} \quad [\textsc{Sel}]$$

$$\frac{\forall i \in I. \quad \Gamma \vdash P_i \triangleright \Delta, c : T_i \rightsquigarrow \widehat{P}_i}{\Gamma \vdash c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I}) \triangleright \Delta, c : \&(\mathtt{p}, \{l_i : T_i\}_{i\in I}) \rightsquigarrow c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i\in I})} \quad [\textsc{Branch}]$$

$$\frac{\Gamma \vdash e : S \rightsquigarrow \widehat{e} \qquad \Delta \text{ end only}}{\Gamma, X : S\ T \vdash X\langle e, c\rangle \triangleright \Delta, c : T \rightsquigarrow X\langle \widehat{e}, c\rangle} \quad [\textsc{Var}]$$

$$\frac{\Gamma, X : S\ t, a : S \vdash P \triangleright b : T \rightsquigarrow \widehat{P} \qquad \Gamma, X : S\ \mu t.T \vdash Q \triangleright \Delta \rightsquigarrow \widehat{Q} \qquad x \rightsquigarrow a \qquad y \rightsquigarrow b}{\Gamma \vdash \mathrm{def}\ X(x, y) = P\ \mathrm{in}\ Q \triangleright \Delta \rightsquigarrow \mathrm{def}\ X(a, b) = \widehat{P}\ \mathrm{in}\ \widehat{Q}} \quad [\textsc{Def}]$$

FIGURE 5.10: Typing and translation rules for MPIM processes

$$\lceil S \rceil = \begin{cases} \lceil G \rceil_{mp(G)} & \text{if } S = G \\ \lceil T \rceil & \text{if } S = T \\ S & \text{otherwise} \end{cases}$$

$$\lceil T \rceil = \begin{cases} !p.\lceil U \rceil.\lceil S' \rceil & \text{if } T = !p.U.S' \text{ or } !^{\wr}p.U.S' \\ ?(\mathtt{p}, \lceil U \rceil).\lceil S' \rceil & \text{if } T = ?(\mathtt{p}, U).S' \text{ or } ?^{\wr}(\mathtt{p}, U).S' \\ \oplus\langle \mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I} \rangle & \text{if } T = \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I} \rangle \\ \&(\mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I}) & \text{if } T = \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) \\ \mu t.\lceil T' \rceil & \text{if } T = \mu t.T' \\ T & \text{otherwise} \end{cases}$$

$$\lceil G \rceil = \begin{cases} \mathtt{p} \to \mathtt{q} : \langle \lceil U \rceil \rangle.\lceil G \rceil & \text{if } G = \mathtt{p} \to \mathtt{q} : \langle U \rangle.G \text{ or } \mathtt{p} \to \mathtt{q} :^{\wr}\langle U \rangle.G \\ \mathtt{p} \to \mathtt{q} : \{l_i : (\lceil G \rceil)_i\}_{i \in I} & \text{if } G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \\ \mu t.\lceil G \rceil & \text{if } G = \mu t.G \\ G & \text{otherwise} \end{cases}$$

FIGURE 5.11: Translation of MPIM types

Chapter 6

Dependent Object Types with Implicit Functions[1]

6.1 Introduction

Implicits are widely used in Scala. They are valuable to programmers as a mechanism for passing context without excessive verbosity. The *Dotty* Scala compiler [Odersky, 2017], written in Scala, contains at more than 5000 occurrences of the `implicit` keyword. Akka [Haller, 2012], a popular Scala library for Actor-based concurrency, uses implicits at the core of its API. Therefore it is of importance to the Scala community to have a solid theoretical foundation for the correctness of Scala's implicit program constructs. Indeed, the safety of implicit functions in Scala has been evidenced by the type-safe integration of implicit functions into lambda calculus [Odersky et al., 2018]. This evidence could be further strengthened by their type-safe integration into DOT, the calculus on which the Dotty compiler is based. In the remainder of this chapter, we present DIF, a type-safe integration of implicit functions into DOT. DIF is shown to be type-safe by translation into the DOT calculus presented in [Amin et al., 2016]. We demonstrate that the type class pattern [Oliveira, Moors, and Odersky, 2010], a typical use case of implicit functions in Scala, can be translated typably into DIF.

Example. The type class pattern [Odersky et al., 2018] in DIF maps closely to the type class pattern as used in Scala. The following example leverages DOT's path-dependency, which exhibits parametric polymorphism by passing objects with abstract type members. The dictionary passing of languages with type classes can then be implemented with implicit functions, achieving ad-hoc polymorphism. Example 1 shows the type class pattern in Scala, and example 2 shows the type class pattern adapted for DIF. In example 1, lines 1–3 represent the class definition - we declare a type class `Ord` with a single definition, `compare`, which compares two values of type `A`. Line 5 represents a type

[1] This chapter is adapted from [Jeffery, 2019], published original work.

class constrained polymorphic function definition, like `print` in our earlier example. Lines 7-9 are is instance definition for `Int`, and line 11 shows an example call to a type class constrained polymorphic function. Example 2 encodes the class declaration across lines 1-5. In place of the type variable in the trait declaration, the DIF encoding includes an abstract type member `A`. The encoding of the `comp` function includes an additional argument over the Scala version. This additional argument is an object with an abstract type member, which the subsequent arguments can use as their type, since DIF does not have type variables. This additional argument is not to be confused with the dictionary argument `ev`, which both encodings require. Example 2 encodes the instance declaration across lines 9-13. We assign the instance to the implicit variable so that it can be passed implicitly to calls to `comp`, and leverage intersection types to make it a subtype of `Ord`. We specify that `A` is restricted to `Int`, and provide a definition of `compare`. Line 16 shows an example call.

```
1   trait Ord[A] {
2       def compare(x: A, y: A): Boolean
3   }
4
5   def comp[A](x: A, y: A)(implicit ev: Ord[A]): Boolean = ev.compare(x, y)
6
7   implicit def intOrd: Ord[Int] = new Ord[Int] {
8       def compare(x: Int, y: Int): Boolean = a < b
9   }
10  ...
11  comp(1, 2)
12  ...
```

Example 1. The type class pattern in Scala

```
1   let ord_package = ν(ord_p) {
2       Ord = μ(self: {
3           A
4           compare: ∀(x: self.A, y: self.A)Boolean
5       })
6       comp: ∀(ty: {A}, x: ty.A, y: ty.A)∀ι(ev: ord_p.Ord∧{A})Boolean =
7           λ(ty: {A}, x: ty.A, y: ty.A)ι.compare(x, y)
8   } in
9   let ι: ord_package.Ord∧{A = Int} = ν(self: {
10          A = Int
11          compare: ∀(x: self.A, y: self.A)Boolean =
12              λ(x: self.A, y: self.A) x < y
13      })
14  in
15  ...
16  ord_package.comp({A}, 1, 2)
17  ...
```

Example 2. The type class pattern in DIF

6.2 The Language DIF

Figure 6.1 gives the syntax of DIF. The syntax of DIF is similar to that of DOT in [Amin et al., 2016], with the addition of constructs related to implicit functions. We add a new type $\forall \wr (u : S) S'$ which represents an implicit (path-dependent) function from S to S'. We also add the *implicit query* $\wr$, the analogue to Scala's `implicitly[T]`. Occurrences of $\wr$ are *resolved* into variables by our typing and translation rules. The variable that replaces a given occurrence of $\wr$ is chosen from candidate implicit variables, which are bound by let constructs, where $\wr$ is the variable name, i.e. **let** $\wr = t$ **in** t'. Again, we highlight additions DIF makes to DOT's syntax in red in the figures in this chapter.

Note that we make a distinction between two sets of variable names. The first, ranged over by x, y is the set of variable names with the implicit query. The second set, ranged over by u, v is the set of variables where $\wr$ is not allowed. At the term level, anywhere we can write a variable name, we could also write an implicit query, so we use x in the grammar of terms. At the type level, however, implicit queries are not allowed in names, and we therefore use u for names in the grammar of types.

TERMS

t	::=	$x.a$	*Selection*
	|	x	*Variable*
	|	$x\ y$	*Application*
	|	**let** $x = t$ **in** t'	*Let-binding*
	|	$\nu(x : S)d$	*Object*
	|	$\lambda(x : S)t$	*Abstraction*

TYPES

S	::=	$\{a : S\}$	*Field Declaration*
	|	$S \wedge S'$	*Intersection*
	|	$\mu(u : S)$	*Recursive type*
	|	$\top$	*Top*
	|	$\bot$	*Bottom*
	|	$\{A : S..S'\}$	*Type Declaration*
	|	$u.A$	*Type Projection*
	|	$\forall(u : S)S'$	*Dependent Function*
	|	$\forall \wr (u : S)S'$	*Implicit Function*

DEFINITIONS

d	::=	$\{A = S\}$	*Type Definition*
	|	$\{a = t\}$	*Field Definition*
	|	$d \wedge d'$	*Aggregate Definition*

VARIABLES (WITH IMPLICIT QUERY)

x, y ::= p | i | $\wr$

VARIABLES

u, v ::= p | i

EXPLICIT VARIABLES $p, q, ...$

IMPLICIT VARIABLES $i, j, ...$

IMPLICIT QUERY $\wr$

TERM MEMBERS $a, b, ...$

TYPE MEMBERS $A, B, ...$

FIGURE 6.1: Grammar of DIF

6.2.1 Abbreviations

We employ the following abbreviations in examples:

– We encode multiple argument function (types) as multiple single function (types):

$$\forall(y : T, z : U) \equiv \forall(y : T)\forall(z : U)$$
$$\lambda(y : T, z : U) \equiv \lambda(y : T)\lambda(z : U)$$

– We group intersection contents together inside { curly brackets }, separating definitions with a newline, additional whitespace or semicolon:

$$\{d \quad d'\} \equiv \{d\} \wedge \{d'\}$$
$$\{d \; ; \; d'\} \equiv \{d\} \wedge \{d'\}$$

– We allow terms in application and selection by encoding them as let-bindings:

$$t\ t' \equiv \textbf{let}\ x = t\ \textbf{in}\ x\ t'$$
$$x\ t \equiv \textbf{let}\ y = t\ \textbf{in}\ x\ y$$
$$t.a \equiv \textbf{let}\ x = t\ \textbf{in}\ x.a$$

– We abbreviate type bounds thusly:

$$A <: T \equiv A : \bot..T \qquad A = T \equiv A : T..T$$
$$A >: T \equiv A : T..\top \qquad A \equiv A : \bot..\top$$

– We encode type ascription as application:

$$t : T \equiv (\lambda(x : T)x)t$$
$$\textbf{let}\ x : T = t\ \textbf{in}\ t' \equiv \textbf{let}\ x = t : T\ \textbf{in}\ t'$$

– Finally we omit types in new-bindings if the type of the definition is explicit, writing $\nu(x : T)d$ as $\nu(x)d$.

6.2.2 Semantics

We define the semantics of DIF by type-based translation from DIF programs to DOT programs. The meaning of a DIF program is therefore given by its typed translation into a DOT program. This translation is the topic of section 6.3. We give the semantics of DOT in figure 6.2.

$$\textbf{let } x = v \textbf{ in } e[x.a] \to \textbf{let } x = v \textbf{ in } e[t] \textbf{ if } v = \nu(x : S)...\{a = t\}...$$
$$\textbf{let } x = v \textbf{ in } e[x\ y] \to \textbf{let } x = v \textbf{ in } e[[z := y]t] \textbf{ if } v = \lambda(z : S)t$$
$$\textbf{let } x = y \textbf{ in } t \to [x := y]t$$
$$\textbf{let } x = \textbf{let } y = t \textbf{ in } u \textbf{ in } v \to \textbf{let } y = t \textbf{ in let } x = u \textbf{ in } v \textbf{ if } y \notin \mathrm{fn}(v)$$
$$e[t] \to e[u] \textbf{ if } t \to u$$

FIGURE 6.2: Semantics of DOT

6.3 Typing for DIF

In this section we introduce the typing system for DIF programs. Typing judgements in DIF are 4-place: $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$. This judgement can be read: under environment Γ, the DIF term t has type S, and translated to the DOT term $\widehat{t}$.

6.3.1 Translation of Types

We define the function $(\bullet)^*$, which translates DIF types into DOT types. DIF types differ from DOT types only in the inclusion of the type for implicit functions $\forall\wr(u : S)S'$. The translation simply erases occurrences of $\wr$, translating implicit functions into explicit functions.

DEFINITION 10 (Translation of types)**.** Figure 6.3 defines translation from DIF types to DOT types.

$$\begin{aligned} x^* &= x & \qquad \bot^* &= \bot \\ \{a : S\}^* &= \{a : S^*\} & \qquad \{A : S..S'\}^* &= \{A : S^*..S'^*\} \\ (S \wedge S')^* &= S^* \wedge S'^* & \qquad (u.A)^* &= u.A \\ \mu(u : S)^* &= \mu(u : S^*) & \qquad (\forall(u : S)S')^* &= \forall(u : S^*)S'^* \\ \top^* &= \top & \qquad (\forall\wr(u : S)S')^* &= \forall(u : S^*)S'^* \end{aligned}$$

FIGURE 6.3: Translation from DIF types to DOT types

We extend the definition of $(\bullet)^*$ pointwise to environments Γ.

6.3.2 Type substitution

DEFINITION 11 (Name substitution on types)**.** We define capture-avoiding substitution of names within types, written $[u := v]S$, in the usual way.

6.3.3 The functions *depth* and *spec*

The function $\boldsymbol{depth}(\Gamma, i, j)$ decides which variable is more deeply nested in the environment Γ, for purposes of disambiguation of implicit variable selection. It returns -1 if i is more deeply nested, and 1 if j is more deeply nested. Its formal definition is given in figure 6.4.

$$\boldsymbol{depth}(\Gamma, i, j) = \begin{cases} -1 & \textbf{if } \Gamma = \Gamma_1, i : S_1, \Gamma_2, j : S_2, \Gamma_3 \\ & \wedge \{i, j\} \cap \textbf{dom}(\Gamma_1 \cup \Gamma_2 \cup \Gamma_3) \neq \emptyset \\ 1 & \textbf{if } \Gamma = \Gamma_1, j : S_1, \Gamma_2, i : S_2, \Gamma_3 \\ & \wedge \{i, j\} \cap \textbf{dom}(\Gamma_1 \cup \Gamma_2 \cup \Gamma_3) \neq \emptyset \\ \bot & \textbf{otherwise} \end{cases}$$

FIGURE 6.4: The ***depth*** function

The function $\boldsymbol{spec}(\Gamma, i, j)$ decides which variable's type is more specific in the environment Γ, i.e. which type can be instantiated to the other by widening or polymorphic parameter instantiation. It returns -1 if i is more specific, 1 if j is more specific, and if they are equal (i.e. both a subtype and a supertype of each other), 0 is returned. Its formal definition is given in figure 6.5.

$$\boldsymbol{spec}(\Gamma, i, j) = \begin{cases} -1 & \textbf{if } \Gamma \vdash i <: j \wedge \neg(\Gamma \vdash j <: i) \\ 1 & \textbf{if } \neg(\Gamma \vdash i <: j) \wedge \Gamma \vdash j <: i \\ 0 & \textbf{if } \Gamma \vdash i <: j \wedge \Gamma \vdash j <: i \\ \bot & \textbf{otherwise} \end{cases}$$

FIGURE 6.5: The ***spec*** function

6.3.4 Typing rules

Figure 6.6 gives the binding rules for DIF binders. DIF binding rules are of the form $x \rightsquigarrow u$, and function in the same way as PIIM's binding rules. DIF's rule [BIND-IM] corresponds to PIIM's rule [T-IMBIND], and DIF's rule [BIND-EX] to PIIM's rule [T-EXBIND].

$$p \rightsquigarrow p \quad [\text{BIND-EX}] \qquad \frac{i \text{ fresh}}{\wr \rightsquigarrow i} \quad [\text{BIND-IM}]$$

FIGURE 6.6: Binding rules for DIF terms

Figure 6.7 gives the binding rules for DIF terms. Most rules are identical to their corresponding rules in DOT (specifically in [Amin et al., 2016]), when ignoring the translation

part of typing judgements ($\rightsquigarrow \widehat{t}$). The typing rules differ slightly from their DOT counterparts in that they have binding judgements in their premises. The rules [VAR-EX], [ALL-EX-I] and [ALL-EX-E] are analogous to the rules $[\text{VAR}]_{DOT}$, $[\text{ALL-I}]_{DOT}$ and $[\text{ALL-E}]_{DOT}$ respectively. The rule [ALL-IM-I] types introduction of implicit functions, introducing an (explicit) abstraction to the translation. The rule [ALL-IM-E] types implicit function elimination, inserting an implicit variable of suitable type as an argument to the implicit function. Finally [VAR-IM] types implicit query, replacing $\wr$ with a chosen implicit variable in the translation. This rule also performs *disambiguation*, whenever more than one implicit variable is of suitable type. Disambiguation follows the same process as Scala (as specified in [Odersky et al., 2018]). We prefer more deeply nested implicit variables over less deeply nested ones, and implicit variables with more specific types over ones with more general types. Where we have a choice between two variables, one of which is more deeply nested and the other a more specific type, we reject the program as ambiguous.

$$\frac{\begin{array}{c} i \in \mathbf{dom}(\Gamma) \\ \forall j \in \mathbf{dom}(\Gamma), j \neq i . \Gamma \vdash \wr : S \rightsquigarrow j \Rightarrow \boldsymbol{depth}(\Gamma, i, j) + \boldsymbol{spec}(\Gamma, i, j) < 0 \end{array}}{\Gamma \vdash \wr : S \rightsquigarrow i} \quad [\text{VAR-IM}]$$

$$\Gamma, p : S, \Gamma' \vdash p : S \rightsquigarrow p \quad [\text{VAR-EX}] \qquad \frac{\Gamma \vdash t : S \rightsquigarrow \widehat{t} \quad \Gamma \vdash S <: S'}{\Gamma \vdash t : S' \rightsquigarrow \widehat{t}} \quad [\text{SUB}]$$

$$\frac{\Gamma \vdash x : S \rightsquigarrow \widehat{x} \quad \Gamma \vdash x : U \rightsquigarrow \widehat{x}}{\Gamma \vdash x : S \wedge U \rightsquigarrow \widehat{x}} \quad [\text{AND-I}]$$

$$\frac{x \rightsquigarrow u \quad \Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t} \quad \{x, u\} \cap \text{fv}(S) = \emptyset}{\Gamma \vdash \lambda(x : S)t : \forall(u : S)U \rightsquigarrow \lambda(u : S^*)\widehat{t}} \quad [\text{ALL-EX-I}]$$

$$\frac{\Gamma \vdash x : \forall(u : S)U \rightsquigarrow \widehat{x} \quad \Gamma \vdash y : S \rightsquigarrow \widehat{y}}{\Gamma \vdash x\, y : [u := y]U \rightsquigarrow \widehat{x}\, \widehat{y}} \quad [\text{ALL-EX-E}]$$

$$\frac{x \rightsquigarrow u \quad \Gamma \vdash t : S \rightsquigarrow \widehat{t} \quad \Gamma, u : S \vdash t' : U \rightsquigarrow \widehat{t'} \quad x \notin \text{fv}(U)}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ t' : U \rightsquigarrow \textbf{let}\ u = \widehat{t}\ \textbf{in}\ \widehat{t'}} \quad [\text{LET}]$$

$$\frac{\Gamma \vdash x : S \rightsquigarrow \widehat{x}}{\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}} \quad [\text{REC-I}]$$

$$\frac{x \rightsquigarrow u \quad \Gamma, u : S \vdash d : S \rightsquigarrow \widehat{d}}{\Gamma \vdash \nu(x : S)d : \mu(u : S) \rightsquigarrow \nu(u : S^*)\widehat{d}} \quad [\{\}\text{-I}]$$

$$\frac{\Gamma \vdash x : \{a : S\} \rightsquigarrow \widehat{x}}{\Gamma \vdash x.a : S \rightsquigarrow \widehat{x}.a} \quad [\text{FLD-E}] \qquad \frac{\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}}{\Gamma \vdash x : S \rightsquigarrow \widehat{x}} \quad [\text{REC-E}]$$

$$\frac{u \text{ fresh} \quad \Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t} \quad u \notin \text{fv}(S)}{\Gamma \vdash t : \forall\wr(u : S)U \rightsquigarrow \lambda(u : S^*)\widehat{t}} \quad [\text{ALL-IM-I}]$$

$$\frac{\Gamma \vdash x : \forall\wr(u : S)U \rightsquigarrow \widehat{x} \quad \Gamma \vdash \wr : S \rightsquigarrow v}{\Gamma \vdash x : [u := v]U \rightsquigarrow \widehat{x}\, v} \quad [\text{ALL-IM-E}]$$

FIGURE 6.7: Typing and translation rules for DIF terms

Figure 6.8 gives the typing rules for DIF definitions. These are again very similar to the typing rules for DOT defintions, and in each case the translations are homomorphic.

$$\Gamma \vdash \{A = S\} : \{A : S..S\} \rightsquigarrow \{A = S^*\} \quad [\text{TYP-I}]$$

$$\frac{\Gamma \vdash t : S \rightsquigarrow \widehat{t}}{\Gamma \vdash \{a = t\} : \{a : S\} \rightsquigarrow \{a = \widehat{t}\}} \quad [\text{FLD-I}]$$

$$\frac{\Gamma \vdash d_1 : S_1 \rightsquigarrow \widehat{d_1} \quad \Gamma \vdash d_2 : S_2 \rightsquigarrow \widehat{d_2} \quad \mathbf{dom}(d_1) \cap \mathbf{dom}(d_2) = \emptyset}{\Gamma \vdash d_1 \wedge d_2 : S_1 \wedge S_2 \rightsquigarrow \widehat{d_1} \wedge \widehat{d_2}} \quad [\text{ANDDEF-I}]$$

FIGURE 6.8: Typing and translation rules for DIF definitions

Figure 6.9 gives the subtyping rules for DIF terms and definitions. These are yet again very similar to the subtyping rules in DOT. The exception is the rule [ALL-<:-ALL-IM], which makes explicit functions a subtype of implicit functions. This allows the passing of arguments explicitly to an implicit function, which is an important aspect of implicit functions in Scala, as it allows overriding as necessary any implicit variable the compiler would otherwise insert.

$$\Gamma \vdash S <: \top \quad [\text{TOP}] \qquad \Gamma \vdash \bot <: S \quad [\text{BOT}] \qquad \Gamma \vdash S <: S \quad [\text{REFL}]$$

$$\frac{\Gamma \vdash S <: U \quad \Gamma \vdash U <: R}{\Gamma \vdash S <: R} \quad [\text{TRANS}] \qquad \frac{\Gamma \vdash S <: U \quad \Gamma \vdash S <: R}{\Gamma \vdash S <: U \wedge R} \quad [<:\text{-AND}]$$

$$\frac{\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u}{\Gamma \vdash S <: u.A} \quad [<:\text{-SEL}] \qquad \frac{\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u}{\Gamma \vdash u.A <: U} \quad [\text{SEL-}<:]$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash \forall (x : S_1) U_1 <: \forall (x : S_2) U_2} \quad [\text{ALL-<:-ALL-EX}]$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash \{A : S_1..U_1\} <: \{A : S_2..U_2\}} \quad [\text{TYP-<:-TYP}]$$

$$\Gamma \vdash S \wedge U <: S \quad [\text{AND}_1\text{-}<:] \qquad \Gamma \vdash S \wedge U <: U \quad [\text{AND}_2\text{-}<:]$$

$$\frac{\Gamma \vdash S <: U}{\Gamma \vdash \{a : S\} <: \{a : U\}} \quad [\text{FLD-<:-FLD}]$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash \forall\!\wr (x : S_1) U_1 <: \forall\!\wr (x : S_2) U_2} \quad [\text{ALL-<:-ALL-IM}]$$

FIGURE 6.9: Subtyping rules for DIF

6.4 Type safety of DIF

As with previous calculi, we show the type safety of DIF by translation into its base calculus, in this case DOT. Given an environment Γ, a DIF term t, a type S and a DOT term $\widehat{t}$, we show that the judgement $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$ implies the judgement $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$. In other words, if a DIF term is typable under Γ with type S, and translates to $\widehat{t}$, then $\widehat{t}$ is typable in DOT under the translation of Γ with the translation of the type S. The same property also holds for definitions, i.e. $\Gamma \vdash d : S \rightsquigarrow \widehat{d} \Rightarrow \Gamma^* \vdash_{DOT} \widehat{d} : S^*$. This property is called type preserving translation. The type safety of DIF depends upon the type safety of DOT, via type preserving translation.

Theorems 9 and 10 show type preserving translation for terms and definitions respectively. We also include some auxiliary lemmas, such as lemma 17, which shows that subtyping is preserved by the translation from DIF types to DOT types.

We show that our translation is type-preserving, and therefore yields a typable DOT term. When resolving variables, insertion of any variable of appropriate type in place of implicit queries is enough to guarantee type-preservation. This means that our soundness result does not address the correctness of our approach to variable selection. We have attempted to design a system that will select the same variable that an analogous Scala program would select, but do not verify this property, and indeed cannot, since Scala's rules about variable selection are not formalised.

Lemma 17 (Preservation of subtyping under type translation)**.** If $\Gamma \vdash S <: U$ then $\Gamma^* \vdash_{DOT} S^* <: U^*$.

Proof. By induction on subtyping derivations.

- **Case** [Top]: $\Gamma \vdash S <: \top$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: \top^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} S^* <: \top$
  - The goal follows from $[\text{Top}]_{DOT}$.
- **Case** [Bot]: $\Gamma \vdash \bot <: S$
  - To show: $\Gamma^* \vdash_{DOT} \bot^* <: S^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \bot <: S^*$
  - The goal follows from $[\text{Bot}]_{DOT}$.
- **Case** [Refl]: $\Gamma \vdash S <: S$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: S^*$
  - The goal is immediate from $[\text{Refl}]_{DOT}$.

- **Case** [TRANS]: $\Gamma \vdash S <: R$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: R^*$
  - By inversion of [TRANS]:
    - $\Gamma \vdash S <: U$
      - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: U^*$
    - $\Gamma \vdash U <: R$
      - Then by induction: $\Gamma^* \vdash_{DOT} U^* <: R^*$
  - The goal then follows from $[\text{TRANS}]_{DOT}$.
- **Case** [<:-AND]: $\Gamma \vdash S <: U \wedge R$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: (U \wedge R)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} S^* <: U^* \wedge R^*$
  - By inversion of [<:-AND]:
    - $\Gamma \vdash S <: U$
      - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: U^*$
    - $\Gamma \vdash S <: R$
      - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: R^*$
  - The goal then holds by $[\text{<:-AND}]_{DOT}$.
- **Case** [<:-SEL]: $\Gamma \vdash S <: u.A$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: (u.A)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} S^* <: u.A^*$
  - By inversion of [<:-SEL]: $\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u$
    - Then by theorem 9: $\Gamma^* \vdash_{DOT} u : \{A : S..U\}^*$
    - Then by definition 10: $\Gamma^* \vdash_{DOT} u : \{A : S^*..U^*\}$
  - The goal then follows from $[\text{<:-SEL}]_{DOT}$.
- **Case** [SEL-<:]: $\Gamma \vdash u.A <: U$
  - To show: $\Gamma^* \vdash_{DOT} (u.A)^* <: U^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} u.A^* <: U^*$
  - By inversion of [SEL-<:]: $\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u$
    - Then by theorem 9: $\Gamma^* \vdash_{DOT} u : \{A : S..U\}^*$
    - Then by definition 10: $\Gamma^* \vdash_{DOT} u : \{A : S^*..U^*\}$
  - The goal then follows from $[\text{SEL-<:}]_{DOT}$.

- **Case** [ALL-<:-ALL-EX]: $\Gamma \vdash \forall(x : S_1)U_1 <: \forall(x : S_2)U_2$
  - To show: $\Gamma^* \vdash_{DOT} (\forall(x : S_1)U_1)^* <: (\forall(x : S_2)U_2)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \forall(x : S_1^*)U_1^* <: \forall(x : S_2^*)U_2^*$
  - By inversion of [ALL-<:-ALL-EX]:
    - $\Gamma \vdash S_2 <: S_1$
      - Then by induction: $\Gamma^* \vdash_{DOT} S_2^* <: S_1^*$
    - $\Gamma, x : S_2 \vdash U_1 <: U_2$
      - Then by induction: $(\Gamma, x : S_2)^* \vdash_{DOT} U_1^* <: U_2^*$
      - And by definition 10: $\Gamma^*, x : S_2^* \vdash_{DOT} U_1^* <: U_2^*$
  - The goal then follows from $[\text{ALL-<:-ALL}]_{DOT}$.
- **Case** [TYP-<:-TYP]: $\Gamma \vdash \{A : S_1..U_1\} <: \{A : S_2..U_2\}$
  - To show: $\Gamma^* \vdash_{DOT} \{A : S_1..U_1\}^* <: \{A : S_2..U_2\}^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \{A : S_1^*..U_1^*\} <: \{A : S_2^*..U_2^*\}$
  - By inversion of [TYP-<:-TYP]:
    - $\Gamma \vdash S_2 <: S_1$
      - Then by induction: $\Gamma^* \vdash_{DOT} S_2^* <: S_1^*$
    - $\Gamma \vdash U_1 <: U_2$
      - Then by induction: $\Gamma^* \vdash_{DOT} U_1^* <: U_2^*$
  - The goal then follows from $[\text{TYP-<:-TYP}]_{DOT}$.
- **Case** $[\text{AND}_1\text{-<:}]$: $\Gamma \vdash S \wedge U <: S$
  - To show: $\Gamma^* \vdash_{DOT} (S \wedge U)^* <: S^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} S^* \wedge U^* <: S^*$
  - The goal follows from $[\text{AND}_1\text{-<:}]_{DOT}$.
- **Case** $[\text{AND}_2\text{-<:}]$: $\Gamma \vdash S \wedge U <: U$
  - To show: $\Gamma^* \vdash_{DOT} (S \wedge U)^* <: U^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} S^* \wedge U^* <: U^*$
  - The goal follows from $[\text{AND}_2\text{-<:}]_{DOT}$.
- **Case** [FLD-<:-FLD]: $\Gamma \vdash \{a : S\} <: \{a : U\}$
  - To show: $\Gamma^* \vdash_{DOT} \{a : S\}^* <: \{a : U\}^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \{a : S^*\} <: \{a : U^*\}$

  - By inversion of [FLD-<:-FLD]: $\Gamma \vdash S <: U$
    - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: U^*$
  - The goal follows from $[\text{FLD-<:-FLD}]_{DOT}$.

- **Case** [ALL-<:-ALL-IM]: $\Gamma \vdash \forall\iota(x : S_1)U_1 <: \forall\iota(x : S_2)U_2$
  - To show: $\Gamma^* \vdash_{DOT} (\forall\iota(x : S_1)U_1)^* <: (\forall\iota(x : S_2)U_2)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \forall(x : S_1^*)U_1^* <: \forall(x : S_2^*)U_2^*$
  - By inversion of [ALL-<:-ALL-IM]:
    - $\Gamma \vdash S_2 <: S_1$
      - Then by induction: $\Gamma^* \vdash S_2^* <: S_1^*$
    - $\Gamma, x : S_2 \vdash U_1 <: U_2$
      - Then by induction: $(\Gamma, x : S_2)^* \vdash U_1^* <: U_2^*$
      - And by definition 10: $\Gamma^*, x : S_2^* \vdash U_1^* <: U_2^*$
  - The goal then follows from $[\text{ALL-<:-ALL}]_{DOT}$.

□

LEMMA 18 (Preservation of free variables under type translation)**.** For all S, $\text{fv}(S) = \text{fv}(S^*)$.

Proof. Immediate from definition 10. □

LEMMA 19 (Commutativity of type translation and name substitution)**.** For all S, u, v, $[u := v](S^*) = ([u := v]S)^*$.

Proof. By induction on S.

- **Case** $\top$
  - To show: $[u := v](\top^*) = ([u := v]\top)^*$
  - By definition 10 on LHS: $[u := v]\top = ([u := v]\top)^*$
  - By definition 11 on LHS and RHS: $\top = \top^*$
  - By definition 10 on RHS: $\top = \top$

- **Case** $\bot$
  - To show: $[u := v](\bot^*) = ([u := v]\bot)^*$
  - By definition 10 on LHS: $[u := v]\bot = ([u := v]\bot)^*$
  - By definition 11 on LHS and RHS: $\bot = \bot^*$
  - By definition 10 on RHS: $\bot = \bot$

- **Case** w
  - To show: $[u := v](w^*) = ([u := v]w)^*$
  - By definition 10 on LHS: $[u := v]w = ([u := v]w)^*$
  - **Case** $u = w$
    - By definition 11 on LHS and RHS: $v = v^*$
    - By definition 10 on RHS: $v = v$
  - **Case** $u \neq w$
    - By definition 11 on LHS and RHS: $w = w^*$
    - By definition 10 on RHS: $w = w$
- **Case** $\mu(u : S')$
  - To show: $[u := v](\mu(w : S')^*) = ([u := v]\mu(w : S'))^*$
  - By definition 10 on LHS: $[u := v]\mu(w : S'^*) = ([u := v]\mu(w : S'))^*$
  - **Case** $u = w$
    - By definition 11 on LHS and RHS: $\mu(w : S'^*) = \mu(w : S')^*$
    - By definition 10 on RHS: $\mu(w : S'^*) = \mu(w : S'^*)$
  - **Case** $u \neq w$
    - By definition 11 on LHS and RHS: $\mu(w : [u := v](S'^*)) = (\mu(w : [u := v]S'))^*$
    - By definition 10 on RHS: $\mu(w : [u := v](S'^*)) = \mu(w : ([u := v]S')^*)$
    - The goal then follows by induction.
- **Case** $\{A : S'..S''\}$
  - To show: $[u := v](\{A : S'..S''\}^*) = ([u := v]\{A : S'..S''\})^*$
  - By definition 10 on LHS: $[u := v](\{A : S'^*..S''^*\}) = ([u := v]\{A : S'..S''\})^*$
  - By definition 11 on LHS and RHS:
    $\{A : ([u := v](S'^*))..([u := v](S''^*))\} = \{A : [u := v]S'..[u := v]S''\}^*$
  - By definition 10 on RHS:
    $\{A : ([u := v](S'^*))..([u := v](S''^*))\} = \{A : ([u := v]S')^*..([u := v]S'')^*\}$
  - The goal then follows by induction.
- **Case** $\{a : S'\}$
  - To show: $[u := v](\{a : S'\}^*) = ([u := v]\{a : S'\})^*$
  - By definition 10 on LHS: $[u := v]\{a : S'^*\} = ([u := v]\{a : S'\})^*$

  - By definition 11 on LHS and RHS: $\{a : [u := v](S'^*)\} = (\{a : [u := v]S'\})^*$
  - By definition 10 on RHS: $\{a : [u := v](S'^*)\} = \{a : ([u := v]S')^*\}$
  - The goal then follows by induction.

- **Case** $u.A$
  - To show: $[u := v]((w.A)^*) = ([u := v](w.A))^*$
  - By definition 10 on LHS: $[u := v](w.A) = ([u := v](w.A))^*$
  - **Case** $u = w$
    - By definition 11 on LHS and RHS: $v.A = (v.A)^*$
    - By definition 10 on RHS: $v.A = v.A$
  - **Case** $u \neq w$
    - By definition 11 on LHS and RHS: $w.A = (w.A)^*$
    - By definition 10 on RHS: $w.A = w.A$

- **Case** $S' \wedge S''$
  - To show: $[u := v]((S' \wedge S'')^*) = ([u := v](S' \wedge S''))^*$
  - By definition 10 on LHS: $[u := v](S'^* \wedge S''^*) = ([u := v](S' \wedge S''))^*$
  - By definition 11 on LHS and RHS:
    $([u := v](S'^*)) \wedge ([u := v](S''^*)) = (([u := v]S') \wedge ([u := v]S''))^*$
  - By definition 10 on RHS:
    $([u := v](S'^*)) \wedge ([u := v](S''^*)) = ([u := v]S')^* \wedge ([u := v]S'')^*$
  - The goal then follows by induction.

- **Case** $\forall(w : S')S''$
  - To show: $[u := v]((\forall(w : S')S'')^*) = ([u := v](\forall(w : S')S''))^*$
  - By definition 10 on LHS: $[u := v](\forall(w : S'^*)S''^*) = ([u := v](\forall(w : S')S''))^*$
  - **Case** $u = w$
    - By definition 11 on LHS and RHS: $\forall(w : S'^*)[u := v](S''^*) = (\forall(w : S')[u := v]S'')^*$
    - By definition 10 on RHS:
      $\forall(w : S'^*)[u := v](S''^*) = \forall(w : S'^*)(([u := v]S'')^*)$
    - The goal then follows by induction.
  - **Case** $u \neq w$
    - By definition 11 on LHS and RHS:
      $\forall(w : [u := v](S'^*))[u := v](S''^*) = (\forall(w : [u := v]S')[u := v]S'')^*$

  - By definition 10 on RHS:
    $\forall(w : [u := v](S'^*))[u := v](S''^*) = \forall(w : ([u := v]S')^*)(([u := v]S'')^*)$
  - The goal then follows by induction.

- **Case** $\forall\wr(u : S')S''$
  - To show: $[u := v]((\forall\wr(u : S')S'')^*) = ([u := v](\forall\wr(u : S')S''))^*$
  - By definition 10 on LHS: $[u := v](\forall(w : S'^*)S''^*) = ([u := v](\forall\wr(w : S')S''))^*$
  - **Case** $u = w$
    - By definition 11 on LHS and RHS: $\forall(w : S'^*)[u := v](S''^*) = (\forall\wr(w : S')[u := v]S'')^*$
    - By definition 10 on RHS:
      $\forall(w : S'^*)[u := v](S''^*) = \forall(w : S'^*)(([u := v]S'')^*)$
    - The goal then follows by induction.
  - **Case** $u \neq w$
    - By definition 11 on LHS and RHS:
      $\forall(w : [u := v](S'^*))[u := v](S''^*) = (\forall\wr(w : [u := v]S')[u := v]S'')^*$
    - By definition 10 on RHS:
      $\forall(w : [u := v](S'^*))[u := v](S''^*) = \forall(w : ([u := v]S')^*)(([u := v]S'')^*)$
    - The goal then follows by induction.

□

THEOREM 9 (Type-preserving translation of DIF terms)**.** If $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$ then $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$.

Proof. By induction on typing derivations.

- **Case** [VAR-EX]: $\Gamma, p : S, \Gamma' \vdash p : S \rightsquigarrow p$
  - To show: $(\Gamma, p : S, \Gamma')^* \vdash_{DOT} p : S^*$
    - Or by definition 10, $\Gamma^*, p : S^*, \Gamma'^* \vdash_{DOT} p : S^*$
  - The goal follows immediately from $[\text{VAR}]_{DOT}$.
- **Case** [VAR-IM]: $\Gamma, i : S, \Gamma' \vdash \wr : S \rightsquigarrow i$
  - To show: $(\Gamma, i : S, \Gamma')^* \vdash_{DOT} i : S^*$
    - Or by definition 10, $\Gamma^*, i : S^*, \Gamma'^* \vdash_{DOT} i : S^*$
  - The goal follows immediately from $[\text{VAR}]_{DOT}$.
- **Case** [SUB]: $\Gamma \vdash t : S' \rightsquigarrow \widehat{t}$

  - By inversion of [SUB]:
    - $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$
      - And by induction: $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$
    - $\Gamma \vdash S <: S'$
      - And by lemma 17: $\Gamma^* \vdash_{DOT} S^* <: S'^*$
  - The goal then follows from $[\text{SUB}]_{DOT}$.

- **Case** [ALL-EX-I]: $\Gamma \vdash \lambda(x : S)t : \forall(u : S)U \rightsquigarrow \lambda(u : S^*)\widehat{t}$
  - To show: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : (\forall(u : S)U)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : \forall(u : S^*)U^*$
  - By inversion of [ALL-EX-I]:
    - $x \rightsquigarrow u$
      - Then either $x = u$ or u fresh
    - $\Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t}$
      - Then by induction: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{t} : U^*$
      - And by definition 10: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{t} : U^*$
    - $\{x, u\} \cap \text{fv}(S) = \emptyset$
      - Then by lemma 18, $\{x, u\} \cap \text{fv}(S^*) = \emptyset$, and then $u \notin \text{fv}(S^*)$
  - The goal then follows from $[\text{ALL-I}]_{DOT}$.

- **Case** [ALL-EX-E]: $\Gamma \vdash x\ y : [u := y]U \rightsquigarrow \widehat{x}\,\widehat{y}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x}\,\widehat{y} : ([u := y]U)^*$
    - Or by lemma 19: $\Gamma^* \vdash_{DOT} \widehat{x}\,\widehat{y} : [u := y]U^*$
  - By inversion of [ALL-EX-E]:
    - $\Gamma \vdash x : \forall(u : S)U \rightsquigarrow \widehat{x}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : (\forall(u : S)U)^*$
      - Then by definition 10: $\Gamma^* \vdash_{DOT} \widehat{x} : \forall(u : S^*)U^*$
    - $\Gamma \vdash y : S \rightsquigarrow \widehat{y}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{y} : S^*$
  - The goal then follows from $[\text{ALL-E}]_{DOT}$.

- **Case** [LET]: $\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ t' : U \rightsquigarrow \textbf{let}\ u = \widehat{t}\ \textbf{in}\ \widehat{t'}$
  - To show: $\Gamma^* \vdash_{DOT} \textbf{let}\ u = \widehat{t}\ \textbf{in}\ \widehat{t'} : U^*$
  - By inversion of [LET]:

  - $x \rightsquigarrow u$
  - $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$
  - $\Gamma, u : S \vdash t' : U \rightsquigarrow \widehat{t'}$
    - Then by induction: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{t'} : U^*$
    - Then by definition 10: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{t'} : U^*$
  - $x \notin \mathrm{fv}(U)$
    - Then by lemma 18: $x \notin \mathrm{fv}(U^*)$
- The goal then follows from $[\textsc{Let}]_{DOT}$.

- **Case** $[\textsc{And-I}]$: $\Gamma \vdash x : S \wedge U \rightsquigarrow \widehat{x}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x} : (S \wedge U)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \widehat{x} : S^* \wedge U^*$
  - By inversion of $[\textsc{And-I}]$:
    - $\Gamma \vdash x : S \rightsquigarrow \widehat{x}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : S^*$
    - $\Gamma \vdash x : U \rightsquigarrow \widehat{x}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : U^*$
  - The goal then follows from $[\textsc{And-I}]_{DOT}$.

- **Case** $[\{\}\text{-I}]$: $\Gamma \vdash \nu(x : S)d : \mu(u : S) \rightsquigarrow \nu(u : S^*)\widehat{d}$
  - To show: $\Gamma^* \vdash_{DOT} \nu(u : S^*)\widehat{d} : (\mu(u : S))^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \nu(u : S^*)\widehat{d} : \mu(u : S^*)$
  - By inversion of $[\{\}\text{-I}]$:
    - $x \rightsquigarrow u$
    - $\Gamma, u : S \vdash d : S \rightsquigarrow \widehat{d}$
      - Then by theorem 10: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{d} : S^*$
      - And by definition 10: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{d} : S^*$
  - The goal then follows from $[\{\}\text{-I}]_{DOT}$.

- **Case** $[\textsc{Fld-E}]$: $\Gamma \vdash x.a : S \rightsquigarrow \widehat{x}.a$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x}.a : S^*$
  - By inversion of $[\textsc{Fld-E}]$: $\Gamma \vdash x : \{a : S\} \rightsquigarrow \widehat{x}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : \{a : S\}^*$

  - And by definition 10: $\Gamma^* \vdash_{DOT} \widehat{x} : \{a : S^*\}$
- The goal then follows from [FLD-E]$_{DOT}$.

- **Case** [REC-I]: $\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S^*)$
  - By inversion of [REC-I]: $\Gamma \vdash x : S \rightsquigarrow \widehat{x}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : S^*$
  - The goal then follows from [REC-I]$_{DOT}$.

- **Case** [REC-E]: $\Gamma \vdash x : S \rightsquigarrow \widehat{x}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x} : S^*$
  - By inversion of [REC-E]: $\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S)^*$
    - And by definition 10: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S^*)$
  - The goal then follows from [REC-E]$_{DOT}$.

- **Case** [ALL-IM-I]: $\Gamma \vdash t : \forall \wr (u : S) U \rightsquigarrow \lambda(u : S^*)\widehat{t}$
  - To show: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : (\forall \wr (u : S)U)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : \forall \wr (u : S^*)U^*$
  - By inversion of [ALL-IM-I]:
    - u fresh
    - $\Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t}$
      - Then by induction: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{t} : U^*$
      - And by definition 10: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{t} : U^*$
    - $u \notin \mathrm{fv}(S)$
      - Then by lemma 18: $u \notin \mathrm{fv}(S^*)$
  - The goal then follows from [ALL-I]$_{DOT}$.

- **Case** [ALL-IM-E]: $\Gamma \vdash x : [u := v]U \rightsquigarrow \widehat{x}\, v$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x}\, v : ([u := v]U)^*$
    - Or by lemma 19: $\Gamma^* \vdash_{DOT} \widehat{x}\, v : [u := v]U^*$
  - By inversion of [ALL-IM-E]:
    - $\Gamma \vdash x : \forall \wr (u : S) U \rightsquigarrow \widehat{x}$

  - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : (\forall \wr (u : S) U)^*$
  - And by definition 10: $\Gamma^* \vdash_{DOT} \widehat{x} : \forall (u : S^*) U^*$
- $\Gamma \vdash \wr : S \rightsquigarrow v$
  - Then by induction: $\Gamma^* \vdash_{DOT} v : S^*$
- The goal then follows from [ALL-E]$_{DOT}$.

□

THEOREM 10 (Type and domain-preserving translation of DIF definitions)**.** If $\Gamma \vdash d : S \rightsquigarrow \widehat{d}$ then $\Gamma^* \vdash_{DOT} \widehat{d} : S^*$ and $\mathbf{dom}(d) = \mathbf{dom}(\widehat{d})$.

Proof. By induction on typing derivations.

- **Case** [TYP-I]: $\Gamma \vdash \{A = S\} : \{A : S..S\} \rightsquigarrow \{A = S^*\}$
  - To show for type preservation: $\Gamma^* \vdash_{DOT} \{A = S^*\} : (\{A : S..S\})^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \{A = S^*\} : \{A : S^*..S^*\}$
  - The goal for type preservation follows immediately from [TYP-I]$_{DOT}$.
  - Domain preservation is immediate from [TYP-I].
- **Case** [FLD-I]: $\Gamma \vdash \{a = t\} : \{a : S\} \rightsquigarrow \{a = \widehat{t}\}$
  - To show for type preservation: $\Gamma^* \vdash_{DOT} \{a = \widehat{t}\} : \{a : S\}^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \{a = \widehat{t}\} : \{a : S^*\}$
  - By inversion of [FLD-I]: $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$
    - Then by theorem 9: $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$
  - The goal for type preservation then follows from [FLD-I]$_{DOT}$.
  - Domain preservation is immediate from [FLD-I].
- **Case** [ANDDEF-I]: $\Gamma \vdash d_1 \wedge d_2 : S_1 \wedge S_2 \rightsquigarrow \widehat{d_1} \wedge \widehat{d_2}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{d_1} \wedge \widehat{d_2} : (S_1 \wedge S_2)^*$
    - Or by definition 10: $\Gamma^* \vdash_{DOT} \widehat{d_1} \wedge \widehat{d_2} : S_1^* \wedge S_2^*$
  - By inversion of [ANDDEF-I]:
    - $\Gamma \vdash d_1 : S_1 \rightsquigarrow \widehat{d_1}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{d_1} : S_1^*$ and $\mathbf{dom}(d_1) = \mathbf{dom}(\widehat{d_1})$
    - $\Gamma \vdash d_2 : S_2 \rightsquigarrow \widehat{d_2}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{d_2} : S_2^*$ and $\mathbf{dom}(d_2) = \mathbf{dom}(\widehat{d_2})$
    - $\mathbf{dom}(d_1) \cap \mathbf{dom}(d_2) = \emptyset$

  - Then $\mathbf{dom}(\widehat{d_1}) \cap \mathbf{dom}(\widehat{d_2}) = \emptyset$ follows since $\mathbf{dom}(d_1) = \mathbf{dom}(\widehat{d_1})$ and $\mathbf{dom}(d_2) = \mathbf{dom}(\widehat{d_2})$
- The goal for type preservation then follows from [AndDef-I].
- Domain preservation holds by induction: if $\mathbf{dom}(d_1) = \mathbf{dom}(\widehat{d_1})$ and $\mathbf{dom}(d_2) = \mathbf{dom}(\widehat{d_2})$ then it follows that $\mathbf{dom}(d_1 \wedge d_2) = \mathbf{dom}(\widehat{d_1} \wedge \widehat{d_2})$.

□

6.5 Conclusion

In this chapter we have shown that implicit functions can be safely integrated into DOT. DOT is a model for the correctness of Scala, and models only a subset of full Scala. We have expanded DOT's coverage of Scala, providing increased confidence that the compiler is bug-free, especially where it pertains to implicit functions. The ability of DIF to express common usage patterns for implicits in Scala shows that our model is faithful to Scala.

In the next and final chapter of this thesis we conclude by looking at our work on implicits critically, considering related work and possibilites for the future study of implicits.

Chapter 7

Conclusion[1]

We have generalised the concept of implicit functions from Scala's sequential setting to message passing concurrency, established the soundness of our proposal by translation, showing its viability in three contexts (IM, PIIM and MPIM) and demonstrated the usefulness of implicit message passing by examples from the literature, including concurrent interpretations of dependency injection and the type class pattern. Our approach to implicit messages in a binary setting generalises straightforwardly to multiparty communication, demonstrating the generality of our approach.

We have shown that implicit functions provide a coherent solution to the repeated rebinding problem of linearly typed languages.

Implicits are useful in sequential programming not just for dependency injection, but also for generic programming [Oliveira et al., 2012]. Our encoding of type classes as sessions leveraging implicits provides evidence that they provide an avenue for further investigation into generic programming for message passing systems, as well as for sequential programming. Such a technology transfer from the domain of sequential to concurrent computation would be aided by a better understanding of the relationship between implicit functions and implicit messages.

The key aim of DOT is to provide a theoretical foundation for Scala. We extend the existing foundation to include implicits, a feature widely used in Scala programming.

Scala's lack of type classes mean that implicits must be leveraged to achieve such behaviour. We have demonstrated the validity of this approach with a typable example in a type-sound calculus.

[1] Portions of this chapter are adapted from [Jeffery and Berger, 2018] and [Jeffery and Berger, 2019], published works co-authored with my supervisor, Dr. Martin BERGER, and [Jeffery, 2019], which is published original work. I estimate that 90% of this chapter is completely my own work, with the last 10% being co-written.

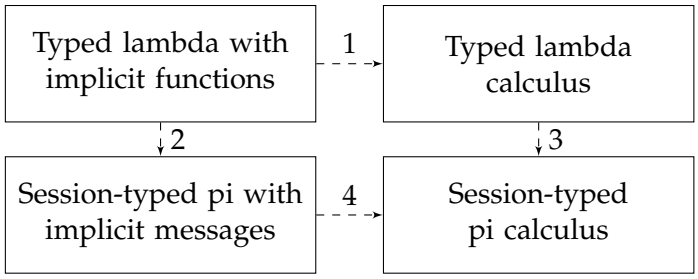


FIGURE 7.1: The relationship between pi and lambda calculi and their extensions with implicit messages

7.1 Further work

7.1.1 Connection between implicit functions and implicit messages

Milner's groundbreaking work on functions as processes [Milner, 1992] gave a deep understanding of lambda calculi as processes engaging only in well-structured interaction. Can Milner's approach clarify the exact correspondence between implicit functions and implicit messages? Precise matches between studied calculi such as, for example, SI [Odersky et al., 2018] and either IM or PIIM are unlikely, because in both cases the calculi are too different: e.g. SI's bidirectional and parametrically polymorphic typing system, which both IM and PIIM lack. PIIM is perhaps closer to SI than IM – certainly it is closer than IM to the simple pi calculus that Milner's translation targets. Clearly choices such as bidirectionality and polymorphism are largely orthogonal to implicits, and we conjecture that full abstraction results between System F and binary session-typed pi calculus [Berger, Honda, and Yoshida, 2005; Toninho and Yoshida, 2017; Giunti and Vasconcelos, 2013] remain stable when source and target calculi are extended with implicits. In order even to be able to state full abstraction we need to generalise the existing equational theories (as well as reasoning tools like typed bisimulations) to lambda and pi calculi with implicits. The nature of any correspondence between a functional language like System F and multiparty session-typed pi calculus is an open question, and thus it is less clear whether such a correspondence could hold in the multiparty context when the source and target calculus are extended with implicits.

Figure 7.1 summarises the conjectured relationship between lambda and pi calculi extended with implicits – boxes labelled with calculi are connected with arrows that connect them via a translation. [Odersky et al., 2018] establishes translation (1), [Toninho and Yoshida, 2017] establishes translation (3), and [Jeffery and Berger, 2018] establishes translation (4). We conjecture that translation (2) is possible, given appropriate lambda and pi calculi. Such a result, completing the square, would lend weight to our claim that implicit messages are the concurrent analogues of implicit functions.

7.1.2 Notions of correctness for implicits

Implicits can be thought of as forms of compile-time metaprogramming. Studies of program equality for compile-time metaprogramming systems have not yet found satisfactory solutions. Notions of equality between metaprograms are required to have strong guarantees about correctness of metaprogramming (soundness, full abstraction), and the same holds for implicits. Indeed for strong correctness results for languages with metaprogramming and implcits, equational theories for their base languages can be leveraged in equational theories extended with metaprogramming and implicits. This is future work.

7.1.3 Ambiguity resolution and coherence

Another open issue is the resolution of ambiguity for implicit message passing. [Odersky et al., 2018] discusses this problem in the context of System F, but modern Scala is based on DOT. DOT's approach to ambiguity follows Scala's as closely as possible, but since there is no formal specification of ambiguity resolution in Scala, no correctness proof for this aspect of DOT was possible. If such a specification is given, a proof becomes possible.

As discussed in section 2.3.5, [Schrijvers, Oliveira, and Wadler, 2017] introduce the concept of coherence for implicit program constructs. Languages with implicit constructs are said to be coherent if they only admit unambiguous programs, that is programs where there is only a single typable choice of variable available for insertion by the compiler. Since we leave implicit variable selection as a nondeterministic choice in IM, PIIM and MPIM, they are clearly not coherent. DIF attempts to model Scala, which is not coherent, so DIF inherits Scala's incoherence. It is not entirely clear if coherence is a desirable property for real programming languages, as such a property might inhibit flexibility and reusability of code. That said, it remains an open question whether it is possible to reformulate the aforementioned systems coherently. The question of coherence for implicit messages is entirely unexplored, and it would be valuable to understand exactly what conditions must be met in order for a program with implicit messages to be coherent – the ambiguity introduced by adjacent implicit receive operations described in section 3.5.1 is clearly an obstacle to coherence and an investigation into mitigation of this is warranted.

7.1.4 Empirical study of implicit messages

This dissertation advertised the utility of implicit messages, but as its argument had to rely on moderately-sized examples, and the aesthetic appeal of smooth generalisation from sequential to concurrent computation. A more substantial empirical evaluation is

desirable. Unfortunately, the well-known difficulties with empirical evaluation of programming languages (see [Kaijanaho, 2015] for an overview) are aggravated here by the absence of mainstream programming language with session-typed message passing concurrency and implicit messages. While [Křikava, Vitek, and Miller, 2019] performs a robust analysis of implicit parameters and conversions, we must leave empirical evaluation of implicit messages as future work.

7.1.5 Type classes and implicits

The well-known correspondence between implicits and type classes [Oliveira, Moors, and Odersky, 2010; Oliveira et al., 2012] would be interesting to establish formally – one can envision a translation $t_{c\to i}$ from type classes to implicits, and given the established results of type classes to lambda calculus $t_{c\to l}$ [Wadler and Blott, 1989] and implicits to lambda calculus $t_{i\to l}$ [Odersky et al., 2018], it seems that it should be possible to establish $t_{c\to i} \cdot t_{i\to l} \equiv t_{c\to l}$ for some notion of $\equiv$.

It has not been investigated whether implicits can be used to encode type classes. One can envison another translation $t_{i\to c}$ from implicits to type classes, and it should then be expected that $t_{i\to c} \cdot t_{c\to i} \equiv t_{c\to i} \cdot t_{i\to c} \equiv \texttt{id}$, again for some $\equiv$.

7.1.6 Type inference

Type inference for HDMP is known to be decidable [Damas and Milner, 1982], and is not broken when extended with type classes [Wadler and Blott, 1989] (at least when heavy handed decisions are made to resolve ambiguity, such as by making instance and class definitions unique and global). Given that there is an apparent equivalence between implicits and type classes, it is likely that type inference for HDMP remains decidable when extended with implicit functions, thought this is yet to be shown, and of course a result formally connecting implicits and type classes could shed light on this. It would likely follow from such a result that the implicit functions in IM do not break decidability of type inference. IM also contains implicit messages, and little is known about the decidability of type inference for implicit messages. The adjacent implicit messages problem would have to be addressed in any type inference algorithm for a language with implicit messages. This means that decidability of type inference for IM, PIIM and MPST are open questions.

It is known that type assignment and subtyping are undecidable for DOT [Rompf and Amin, 2016] since DOT can encode system $F_{<:}$. As DIF is a superset of DOT, these properties hold for DIF transitively. DOT does, however, have a local type inference procedure [Pierce and Turner, 2000]. It follows that such a procedure should also exist for DIF, especially since local type inference is used to great effect in Scala, a language

with implicits. DOT has no principal types, and it is therefore likely that DIF also lacks principal types, although this remains to be shown.

7.1.7 Implicits and concurrency

While Scala concurrency libraries leverage implicits, it is unclear whether their usage is orthogonal to concurrency, or whether there are deeper connections between the two concepts that can be further studied in theory.

7.1.8 Extending DOT to model more of Scala

While our type soundness result for DIF lends plausibility to the correctness (or, the provability of correctness) of Scala's implicits, DIF lacks many features of Scala. As more of Scala's features are modelled in DOT, it becomes possible to further extend DIF to match new features, and show that the new features do not introduce bugs in the presence of implicits [2].

7.1.9 Implicits and linearity

It is clear that linearity provides implicit resolution systems an additional criterion for the choice of variable. Unused implicit linear variables are clear targets for compiler insertion. Conversely, selection of implicits aids in satisfying linearity constraints. This is the case in IM and PIIM. Further study of the connection between linearity and implicits is warranted, and would be of interest to the Scala community, since the addition of linearity to Scala would provide clear benefits. An integration of linear types into DIF would provide a basis for this.

7.1.10 Automatic conversion of explicit parameters to implicit paramters

Implicits allow to hide repetitive argument passing. It is likely possible to identify algorithmically, with the help of heuristics, when a program contains repetition that could be hidden with implicits, and automate the process of adapting source code to make those arguments implicit. A tool implementing this behaviour would likely make a useful addition to development environments for languages with implicit parameters.

[2] Indeed this is already possible as of July 2019 due to [Rapoport and Lhoták, 2019].

7.2 Related Work

7.2.1 Session types

[Arslanagić, Pérez, and Voogd, 2019] introduce the concept of *minimal* session types, which are session types that lack sequencing - instead of the general form $S ::= !T.S | ?T.S | \texttt{end}$, continuations are disallowed, and we have only $S ::= !T.\texttt{end} | ?T.\texttt{end}$. They show that Higher Order (HO) session types can be encoded in Minimal Higher Order (MHO) session types, and provide a sound translation from the former to the latter. Where $\mathcal{G}$ translates types and environments, and $\mathcal{B}$ terms, they show that $\Delta \vdash_{HO} P$ implies $\mathcal{G}(\Delta) \vdash_{MHO} \mathcal{B}(P)$. The idea of translating types and terms separately and analogously, and showing that typability is preserved by the translation is the same idea used to show soundness for our calculi with implicits.

7.2.2 Implicits

[Madsen and Lhoták, 2018] introduce the concepts of implicit *attributes* and *implicified predicates* for logic programming. They develop a calculus Δ_{Dat}, a minimal horn clause based logic calculus similar to Datalog, with these implicit features. A translation is then given from Δ_{Dat} to a logic language that differs from Δ_{Dat} only in that it lacks implicits. It is argued with examples that implicit parameters are useful in logic programming. Their approach is near identical to our approach with DIF, except that the base calculus is a logic language rather than DOT. One limitation of this work are that they do not consider polymorphism but only consider monomorphic types. Their calculus also lacks type constructors.

[Madsen and Lhoták, 2018] also identify 4 criteria that they argue that any reasonable design of implicit parameters should satisfy: Type safety; Consistency - resolution of implicit parameters into explicit ones should be identity for programs with no implicit parameters, and in the case that implicit parameters are always given explicitly at call sites, resolution should simply erase implicit keywords; Determinism - this is the concept of coherence seen in [Schrijvers, Oliveira, and Wadler, 2017]; and Predictability - explicit parameters should not be modified by resolution of implicit ones. As discussed, coherence for our calculi are open problems. We show type safety for all our calculi, but determinism and predictability for them are also open questions.

[Křikava, Vitek, and Miller, 2019] perform the first empirical analysis of the use of implicit parameters and conversions, analysing 6,500 Scala repositories on GitHub including 7.9 million call-sites (almost 25% of all call sites) involving implicits and 900 thousand implicit declarations across 18.5M lines of code. A simple calculation then yields the value $8.\bar{7}$ for the number of calls to every implicit definition. Primarily addressed are questions of pervasiveness, complexity and performance of implicits. 98.7% of code

bases analysed contained at least one implicit call site, and 79.4% of code bases contained an implicit definition, 14.2% of which occurred in tests and the rest in main code. The type class pattern was found to be one of the main applications of implicit parameters.

[Norell, 2007] aims to make programming with dependent types more practical by extending the dependently typed language Agda with pattern matching, modules, and, most relevantly, *metavariables*, which easily allow extension of Agda with implicit parameters. Implicit parameters are resolved in a translation phase.

[Sozeau and Oury, 2008] leverages Agda's implicit parameters to encode type classes in a 'first-class' manner. This adds weight to the conjecture of a strong theoretical connection between implicits and type classes.

[Brady, 2013] details the programming language Idris, a dependently typed language with implicit parameters. Idris allows free type or value variables in type signatures to have their binders and associated types inferred by the compiler. In languages without dependent types, type variable binders can also be inferred, but Idris differs in that the dependent types associated with those binders can also be inferred by the compiler.

[White, Bour, and Yallop, 2015] add *modular* implicits to OCaml, a widely-used, high-level, strongly typed functional programming language. Their modular implicits are modular in the sense that they operate at the module level - OCaml has typed modules, and implicit modules allow for ad-hoc overloading and other related properties at the module level in the same way as implicit parameters allow ad-hoc overloading and related properties at the term level. Their implicit modules resemble Scala's implicit classes.

[Turon, 2017] discusses the trade-offs of implicit program constructs. Implicits are often dismissed for introducing bugs that are hard to track down due to subtle chains of inference. This can indeed be a problem – implicits (and indeed language features in general) should be designed in such a way as to elide repeated and obvious parts of the code, whilst making everything relevant to understand a piece of code easy to see or find. [Turon, 2017] argues that good implicit feature design trades off *applicability* – where implicit constructs can be used; *power* – what information can be hidden and how implicits change the code's typing or behaviour; and *context-dependence* – how elided details are filled in, and the ease with which one can learn or understand what is implicit in a piece of code.

7.2.3 Scala and DOT

[Rapoport and Lhoták, 2019] expand the set of Scala's features modelled by DOT to include Scala's *fully* path-dependent types and *singleton types*. Previous formulations of DOT allowed paths to types of length one only, of the form $u.A$, whereas full Scala allows arbitrary paths, e.g. `core.symbols.Symbol` (an example from Scala's Dotty compiler).

These full paths are required to express type dependencies in the module system. Singleton types are those of the form `p.type`, which is a type equivalent to the type of `P`. More concretely, the type obtained by the expression `1.type` is `Int`. The paper introduces a calculus pDOT that allows full paths and singleton types as in Scala, with a mechanised soundness proof. It remains to be seen whether pDOT can also be extended with implicit functions.

Bibliography

Abadi, Martin and Luca Cardelli (1994). "A Theory of Primitive Objects". In: *European Symposium on Programming*. Springer, pp. 1–25.

Agha, Gul (1986). "An Overview of Actor Languages". In: *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming*. OOPWORK '86. Yorktown Heights, New York, USA: ACM, pp. 58–67. ISBN: 0-89791-205-5. DOI: 10.1145/323779.323743. URL: http://doi.acm.org/10.1145/323779.323743.

Amin, Nada, Adriaan Moors, and Martin Odersky (2012). "Dependent Object Types". In: *19th International Workshop on Foundations of Object-Oriented Languages*. EPFL-CONF-183030.

Amin, Nada, Tiark Rompf, and Martin Odersky (2014). "Foundations of path-dependent types". In: *Acm Sigplan Notices*. Vol. 49. 10. ACM, pp. 233–249.

Amin, Nada et al. (2016). "The Essence of Dependent Object Types". In: *A List of Successes That Can Change the World*. Springer, pp. 249–272.

Arslanagić, Alen, Jorge A Pérez, and Erik Voogd (2019). "Minimal Session Types (Extended Version)". In: *arXiv preprint arXiv:1906.03836*.

Atkey, Robert (2009). "Parameterised Notions of Computation". In: *JFP* 19.3-4, pp. 335–376. ISSN: 0956-7968. DOI: 10.1017/S095679680900728X. URL: http://dx.doi.org/10.1017/S095679680900728X.

Berger, Martin, Kohei Honda, and Nobuko Yoshida (2005). "Genericity and the π-Calculus". In: *Acta Informatica*.

Bernardy, Jean-Philippe et al. (Dec. 2017). "Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language". In: *Proc. ACM Program. Lang.* 2.POPL, 5:1–5:29. ISSN: 2475-1421. URL: http://doi.acm.org/10.1145/3158093.

Brady, Edwin (2013). "Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation". In: *Journal of functional programming* 23.5, pp. 552–593.

Church, Alonzo (1932). "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics*, pp. 346–366. URL: https://www.jstor.org/stable/pdf/1968337.pdf.

— (1940). "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2, pp. 56–68.

Coppo, Mario et al. (2015). "Global Progress in Dynamically Interleaved Multiparty Sessions". In: *Mathematical Structures in Computer Science*.

Damas, Luis and Robin Milner (1982). "Principal Type-Schemes for Functional Programs." In: *POPL*. Vol. 82, pp. 207–212.

Gay, Simon J. and Vasco T. Vasconcelos (2010). "Linear type theory for asynchronous session types". In: *JFP* 20.1, 19–50. DOI: 10.1017/S0956796809990268.

Girard, Jean-Yves (1972). "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Université Paris VII.

— (Jan. 1987). "Linear Logic". In: *Theoretical Computer Science* 50.1, pp. 1–102. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90045-4. URL: http://dx.doi.org/10.1016/0304-3975(87)90045-4.

Giunti, Marco and Vasco Thudichum Vasconcelos (2013). *Linearity, Session Types and the Pi Calculus*.

Haller, Philipp (2012). "On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective". In: *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. ACM, pp. 1–6.

Hindley, Roger (1969). "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146, pp. 29–60.

Hoare, C.A.R. (1985). *Communicatiing Sequential Processes*. Prentice Hall.

Honda, Kohei (1993). "Types for Dyadic Interaction". In: *Proc. CONCUR*.

Honda, Kohei, Vasco T. Vasconcelos, and Makoto Kubo (1998). "Language primitives and type disciplines for structured communication-based programming". In: *Proc. ESOP*. Vol. 1381. LNCS, pp. 22–138.

Honda, Kohei, Nobuko Yoshida, and Mario Carbone (2008). "Multiparty Asynchronous Session Types". In: *Proc. POPL*, pp. 273–284.

Jeffery, Alex (2019). "Dependent Object Types with Implicit Functions". In: *Scala Symposium*.

Jeffery, Alex and Martin Berger (2018). "Asynchronous Sessions with Implicit Functions and Messages". In: International Symposium on Theoretical Aspects of Software Engineering (TASE).

— (2019). "Asynchronous Sessions with Implicit Functions and Messages, Extended Version". In: *Science of Computer Programming* 180, pp. 36–70.

Kaes, Stefan (1988). "Parametric Overloading in Polymorphic Programming Languages". In: *ESOP '88*. Ed. by H. Ganzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 131–144.

Kaijanaho, Antti-Juhani (2015). "Evidence-based programming language design: a philosophical and methodological exploration". PhD thesis. University of Jyväskylä.

Kiselyov, Oleg (2014). *Implementing, and Understanding Type Classes*. https://web.archive.org/web/20180910165920/http://okmij.org/ftp/Computation/typeclass.html.

Knuth, Donald E and Luis Trabb Pardo (1980). "The Early Development of Programming Languages". In: *A history of computing in the twentieth century*. Elsevier, pp. 197–273.

Kobayashi, Naoki, Benjamin C Pierce, and David N Turner (1999). "Linearity and the Pi-Calculus". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.5, pp. 914–947.

Křikava, Filip, Jan Vitek, and Heather Miller (2019). "Scala Implicits are Everywhere". In: *Proc. OOPSLA*.

Lewis, Jeffrey R. et al. (2000). "Implicit Parameters: Dynamic Scoping with Static Types". In: *Proc. POPL*.

Lindley, Sam and J Garrett Morris (2016). "Talking Bananas: Structural Recursion for Session Types". In: *ACM SIGPLAN Notices*. Vol. 51. 9. ACM, pp. 434–447.

Madsen, Magnus and Ondřej Lhoták (2018). "Implicit Parameters for Logic Programming". In: *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany*. ACM. DOI: 10.1145/3236950.3236953.

Martin-Löf, Per (1975). "An Intuitionistic Theory of Types: Predicative Part". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, pp. 73–118.

Milner, Robin (1989). *Communication and Concurrency*. Prentice Hall.

— (1992). "Functions as Processes". In: *MSCS* 2.2, pp. 119–141.

— (1999). *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press.

Milner, Robin and Parrow, Joachim and Walker, David (1992). "A Calculus of Mobile Processes, I". In: *Information and Computation* 100.1, pp. 1 –40. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90008-4. URL: http://www.sciencedirect.com/science/article/pii/0890540192900084.

Nestmann, Uwe and Benjamin C Pierce (1996). "Decoding Choice Encodings". In: *International Conference on Concurrency Theory*. Springer, pp. 179–194.

Norell, Ulf (2007). *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

— (2008). "Dependently typed programming in Agda". In: *International School on Advanced Functional Programming*. Springer, pp. 230–266.

Odersky, Martin (2019). *A Tour of Scala 3*. https://web.archive.org/web/20190908123007/https://www.youtube.com/watch?v=_Rnrx2lo9cw.

Odersky, Martin et al. (2017). *Dotty Compiler: A Next Generation Compiler for Scala*. https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/.

Odersky, Martin et al. (2018). "Simplicitly: Foundations and Applications of Implicit Function Types". In: *Proc. POPL*.

Oliveira, Bruno C. d. S. et al. (2012). "The Implicit Calculus: A New Foundation for Generic programming". In: *ACM SIGPLAN Notices*. Vol. 47. 6. ACM, pp. 35–44.

Oliveira, Bruno C.d.S., Adriaan Moors, and Martin Odersky (2010). "Type Classes As Objects and Implicits". In: *Proc. OOPSLA*, pp. 341–360.

Pierce, Benjamin C. and David N. Turner (Jan. 2000). "Local Type Inference". In: *ACM Trans. Program. Lang. Syst.* 22.1, pp. 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100. URL: http://doi.acm.org/10.1145/345099.345100.

Rapoport, Marianna and Ondřej Lhoták (2019). "A Path To DOT: Formalizing Fully Path-Dependent Types". In: *Proc. OOPSLA*.

Reynolds, John C (1974). "Towards a theory of type structure". In: *Programming Symposium*. Springer, pp. 408–425.

Rice, Henry Gordon (1953). "Classes of Recursively Enumerable Sets and their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2, pp. 358–366.

Rompf, Tiark and Nada Amin (2016). "Type Soundness for Dependent Object Types (DOT)". In: *ACM Sigplan Notices*. Vol. 51. 10. ACM, pp. 624–641.

Russell, Bertrand and Alfred North Whitehead (1910–13). *Principia Mathematica*. URL: https://web.archive.org/web/20181018104855/https://en.wikipedia.org/wiki/Principia_Mathematica.

Schneider, Fred B, Greg Morrisett, and Robert Harper (2001). "A Language-Based Approach to Security". In: *Informatics*. Springer, pp. 86–101.

Schrijvers, Tom, Bruno CdS Oliveira, and Philip Wadler (2017). "Cochis: Deterministic and Coherent Implicits". In: *Report CW* 705.

Sobral, Daniel C. and Matthias Braun (2011). *Where does Scala look for implicits?* https://web.archive.org/web/20181119203031/https://docs.scala-lang.org/tutorials/FAQ/finding-implicits.html.

Sozeau, Matthieu and Nicolas Oury (2008). "First-Class Type Classes". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer, pp. 278–293.

Strachey, Christopher (2000). "Fundamental Concepts in Programming Languages". In: *Higher-order and symbolic computation* 13.1-2, pp. 11–49.

Takeuchi, Kaku, Kohei Honda, and Makoto Kubo (1994). "An Interaction-based Language and its Typing System". In: *Proc. PARLE*.

Toninho, Bernardo and Nobuko Yoshida (2017). "On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings". In: *CoRR* abs/1711.00878. arXiv: 1711.00878. URL: http://arxiv.org/abs/1711.00878.

Turner, David N. (1995). "The Polymorphic Pi-calculus: Theory and Implementation". PhD thesis. University of Edinburgh.

Turon, Aaron (2017). *Rust's language ergonomics initiative*. https://web.archive.org/web/20190824062128/https://blog.rust-lang.org/2017/03/02/lang-ergonomics.html.

Wadler, P. and S. Blott (1989). "How to Make Ad-hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. New York, NY, USA: ACM, pp. 60–76. DOI: 10.1145/75277.75283. URL: http://doi.acm.org/10.1145/75277.75283.

Walker, David (2005). "Substructural type systems". In: *Advanced Topics in Types and Programming Languages*, pp. 3–44.

White, Leo, Frédéric Bour, and Jeremy Yallop (2015). "Modular implicits". In: *arXiv preprint arXiv:1512.01895*.