1 Introduction

1.1 From Data Processing to Communication

For a long time, approximately from the 1950s to the 1990s, most computing consisted of isolated computers doing data processing. The importance of structured data was realised at an early stage, and the first high-level programming languages supported data structures and data types. Figure 1.1 shows examples of data structure declarations with types in Cobol and Fortran, and Figure 1.2 shows examples in the modern languages Rust and Haskell. Niklaus Wirth, the inventor of the programming language Pascal, used the slogan "algorithms + data structures = programs" as the title of a classic textbook, and almost certainly most of the readers of this book have attended or taught a course with a similar title.

Programming languages allow data structures to be codified as data types, and programming tools and environments use data types as the basis for analysis and

Cobol:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NAME    PIC X(25).
01 WS-CLASS   PIC 9(2)  VALUE  '10'.

01 WS-ADDRESS.
   05 WS-HOUSE-NUMBER    PIC 9(3).
   05 WS-STREET          PIC X(15).
   05 WS-CITY            PIC X(15).
   05 WS-COUNTRY         PIC X(15)  VALUE 'INDIA'.
```

Fortran:

```
INTEGER   COLS,ROWS
PARAMETER(ROWS=12,COLS=10)
REAL      MATRIX(ROWS,COLS),VECTOR(ROWS)
```

Figure 1.1 Typed data structure declarations in Cobol (top, from www.tutorialspoint.com/cobol/cobol_data_types.htm) and Fortran (bottom, from www.obliquity.com/computer/fortran/initial.html).

Rust:

```
struct Address {
   houseNumber : u16,
   street : String,
   city : String,
   country : String
}
```

Haskell:

```
data Tree a = Leaf
            | Node a (Tree a) (Tree a)
```

Figure 1.2 Typed data structure declarations in Rust (top) and Haskell (bottom).

verification. This could be at compile time, in languages such as Java, C♯, Scala and Haskell, or at run time, in languages such as Python. In a typical integrated development environment (IDE) such as Eclipse, writing code that applies an operation to the wrong data type (for example, calling a method that doesn't exist in a class) produces a visual highlight of the error (such as a × or an underline). Clicking on the error then produces a menu suggesting operations that can be validly applied.

More recently, the nature of computing has changed. We now depend on systems of communicating programs: web apps and web services, mobile apps and their connections to servers, cloud services, data centres, software architectures based on micro-services, and more. Even a typical device that we think of as a single computer contains a multicore processor, and further speed increases in computing will depend on effective use of increasingly parallel architectures and the communication that they require.

We can update Wirth's slogan to a new slogan: "programs + communication structures = systems". Communication structures are essential for the design of systems, as evidenced by the large number of standard communication protocols that specify the sequence and format of messages in communication-based systems. In order for programming languages and tools to support programmers in the correct implementation of communication, there needs to be a way of codifying communication structures in the same way that data types codify data structures. This is where session types come in.

1.2 From Data Types to Session Types

Session types are type-theoretic specifications of communication protocols, so that protocol implementations can be verified by compile-time type checking in a programming language. We can introduce the key ideas through a series of examples. An extremely simple protocol specifies sending one message, which is an integer, and then closing the connection. The corresponding session type is

$$!\mathsf{int}.\mathsf{end}_!$$

in which ! means send, or output, the dot means sequencing, and $\mathsf{end}_!$ means that the channel must be closed. If there is a software component that implements this protocol, by sending a particular integer and then closing the connection, then there should be another component that interacts with it by receiving an integer and then waiting for the connection to be closed. The protocol followed by the latter component is described by the session type

$$?\mathsf{int}.\mathsf{end}_?$$

in which ? means receive, or input, and $\mathsf{end}_?$ means wait for closure of the connection. These protocols are related by a notion of duality, which swaps sending and receiving steps.

Typical protocols do not only send messages in one direction. The session type

$$?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.\mathsf{end}_?$$

describes a slightly more complex protocol in which two integers are received, one after the other, and then an integer is sent. The dual session type describes the protocol followed by the other component:

$$!\mathsf{int}.!\mathsf{int}.?\mathsf{int}.\mathsf{end}_!$$

Another feature of most protocols is choice. At a certain point in the protocol, perhaps at the beginning, one component offers a choice between several patterns of interaction, and the other component makes a choice. Here is a protocol for a compute service, written to specify the behaviour of the server. It offers a choice (&) between two operations: addition (plus) and negation (neg).

$$\&\{\mathsf{plus}\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.\mathsf{end}_?, \mathsf{neg}\colon ?\mathsf{int}.!\mathsf{int}.\mathsf{end}_?\}$$

Each option is followed by a session type describing the protocol for that operation. If the client chooses plus then the server must receive two integers in sequence and then send an integer which is supposed to be their sum. If the client chooses neg then the server must receive an integer and then send an integer which is supposed to be its negation. This form of choice is called *external* choice because the choice is made by the component at the other endpoint of the channel.

Again we have a dual session type that describes the protocol followed by a client. The new ingredient is the constructor $\oplus$ for making a choice.

$$\oplus\{\mathsf{plus}\colon !\mathsf{int}.!\mathsf{int}.?\mathsf{int}.\mathsf{end}_!, \mathsf{neg}\colon !\mathsf{int}.?\mathsf{int}.\mathsf{end}_!\}$$

The client implements this type by sending as the first message one of the labels plus or neg. After that, it sends and receives messages according to the continuation protocol of the label that it sent. This is an *internal* choice by the client.

We have adopted the convention that the client closes the connection ($\mathsf{end}_!$) and the server waits for it to be closed ($\mathsf{end}_?$).

Realistic protocols usually allow repetition. We can express repetition by introducing recursive session types. For example, the following session type allows the server to offer a sequence of operations, terminating when the client selects quit.

$$S = \&\{\mathsf{plus}\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.S, \mathsf{neg}\colon ?\mathsf{int}.!\mathsf{int}.S, \mathsf{quit}\colon \mathsf{end}_?\}$$

The dual session type, describing the client's protocol, becomes recursive in a corresponding way.

1.3 Assumptions about Communication

The theory of session types makes some assumptions about the underlying communication mechanism that protocols are built on top of. First, we are working in a concurrent or distributed setting, with communication over *channels*. When a message is sent, it is consumed by exactly one receiver. This is in contrast to broadcast communication, for example. We also assume that reliable message delivery is guaranteed, and that the order of messages is preserved. The implementation of TCP/IP sockets in a typical programming language is a good example of what we have in mind. In theoretical accounts of session types, we consider channels to be bidirectional. A bidirectional channel could be implemented as a pair of unidirectional channels. A channel has two *endpoints*, and each message travels from one endpoint to the other. In general there can be multiple references to each endpoint, but messages are not shared. Also, one component in a distributed system can use any number of channels for sending or receiving. This is in contrast to the *actor* paradigm, in which each component has a single mailbox for incoming messages.

Session types can be applied to synchronous and asynchronous models of communication. Synchronous communication, in which the sender and receiver synchronise on every message, is theoretically simpler but is only realistic as a model of local concurrency. Asynchronous communication assumes that messages go into a queue at the receiver, and is more realistic for distributed systems. A formal model of asynchronous communication requires the queues to be considered as part of the semantics. However, from the point of view of type checking programs, there is no difference between synchronous and asynchronous communication.[1] In this book we only consider synchronous communication.

An important aspect of session type systems is control of ownership of channel endpoints. In the client/server example discussed in the previous section, imagine two clients sharing a connection to a single server. If both clients send messages to select a service—for example, one might send add and the other might send neg—the server receives one of the messages first. The other message is then

[1] Unless we consider asynchronous subtyping, which is discussed briefly in Section 5.6.

unexpected, because the server has already entered the protocol for one particular service. This is a race condition which conflicts with the assumption of reliable message delivery, because in effect the label chosen by the second client cannot be delivered. The simplest way to avoid this situation, which is followed by most session type systems, is to specify that each of the two endpoints of a given channel is accessible to exactly one thread at a time. Technically, this is achieved by using concepts from linear type theory. This restriction can be relaxed in the special case of stateless protocols, as we explain in Chapter 4.

1.4 Session Types in Programming Languages

We have introduced the concepts of session types without considering which language they are a type system for. In most of the book we present a session type system for a form of pi calculus, which we regard as a core concurrent programming language. This gives the simplest possible setting in which to define the syntax and operational semantics of a language for which we can then define typing rules for session types. In Chapter 7 we define a core functional language with concurrency and communication features, and define a session type system for it. Session types have also been applied to imperative and object-oriented languages. Generally speaking, the more features a language has, the more complex its session type system becomes, because of the need to consider interactions between session types, linear typing and various other language constructs.

1.5 The Safety Guarantees of Session Types

In this book, we focus on session types as a static type system, so that compile-time type checking guarantees certain run-time behavioural properties. It is also possible to apply session types to dynamically typed languages, by interpreting types as specifications of run-time monitors which detect protocol violations. Working with static session type systems, there are two main guarantees for well-typed programs (or systems of distributed programs). First, no communication mismatch: if the owner of one endpoint of a channel sends a message, then the owner of the other endpoint is expecting to receive a message, and the message types match; conversely if the owner of one endpoint is expecting to receive a message of a certain type, then the owner of the other endpoint will send a message of the expected type. Second, session fidelity: the sequence and types of the messages sent on a channel match the session type of the channel. Formally we define runtime errors to be programs that fail to communicate because they connect an input to an input, or connect an output to a choice, and so on. We then prove that well-typed programs cannot evolve into runtime errors. The higher-level properties, no communication mismatch and session fidelity, emerge from the proof that runtime errors cannot occur. All of these properties can be proved with respect to the formal type system and operational semantics.

Some session type systems guarantee deadlock-freedom. This means that every send operation eventually finds a matching receive, and conversely every input operation eventually finds a matching output. Most of the type systems in this book do not guarantee deadlock-freedom. The exception is the logically-based type system in Chapter 9.

1.6 Binary and Multiparty Session Types

The theory of session types comes in two varieties. The original form, which is now known as *binary* session types, considers every communication channel separately. Every protocol is between the two endpoints of a single channel. It is possible for one component to run several protocols on different channels, with each protocol described by its own session type, but the type system does not specify any relationship between messages in one protocol and messages in another protocol. This gives a relatively simple theory but the behavioural guarantees are rather weak: a system can easily deadlock even if all of its point-to-point protocols are correctly implemented.

Multiparty session types consider collective protocols among groups of components. The theory offers a methodology for the design of communicating systems, in which the first stage is to define a global protocol that specifies all the messages exchanged between a certain set of components. Subject to some consistency conditions, the global type can be projected to a collection of local types, one for each component. A local type is similar to a binary session type, in that it specifies the sequence of communications that one component can do, but it includes communications between one component and all the other components. Local types can be used as the basis for type checking in pi calculus or a programming language. Because all the messages in the system are specified together, including consideration of how they can be interleaved, the behavioural guarantees offered by multiparty session types include strong deadlock-freedom properties.

In this book we only deal with binary session types. Multiparty session types deserve a book of their own. However, a thorough understanding of the theory of binary session types is a good preparation for tackling the literature on multiparty session types.

2 Basic Concepts

We begin by presenting the basic concepts of session types, using the pi calculus as a core concurrent programming language for which we define a type system. We assume some familiarity with the pi calculus and the concepts of operational semantics and type systems. References to background reading can be found at the end of the chapter.

2.1 Session Types

Moving on from the informal description in Chapter 1, we now define session types formally. We begin with finite types built from input, output and the two forms of the terminated type. These are defined by the grammar in Figure 2.1. In this and other figures, we highlight new parts of the definition. As this is the first definition of the syntax of types, every clause is highlighted. We add external and internal choice types later in the chapter (Section 2.5).

A session type describes the communication operations that can be performed on one endpoint of a communication channel. The type $\mathsf{end}_!$ means that communication has finished and the only remaining operation is to close the channel. The type $\mathsf{end}_?$ also means that communication has finished, but the channel will be closed from the other endpoint. The type $?T.U$ means that a message of type T can be received from the channel, and subsequently the channel must be used according to type U. The type $!T.U$ means that a message of type T can be sent on the channel, and subsequently the channel must be used according to type U. We use meta-variables T, U, V, W to denote types.

We often use $\mathsf{end}_!$ and $\mathsf{end}_?$ as illustrative message types. For example, $?\mathsf{end}_!.\mathsf{end}_?$ is a type that describes receiving (?) a message of type $\mathsf{end}_!$ and then waiting for the channel to be closed ($\mathsf{end}_?$). The message received, of type $\mathsf{end}_!$, is a channel endpoint that can be used by closing it. For a more elaborate example, $!(?\mathsf{end}_!.\mathsf{end}_?).\mathsf{end}_!$ is a type that describes receiving a message of type $?\mathsf{end}_!.\mathsf{end}_?$ and then actively closing the channel ($\mathsf{end}_!$).

A key part of the theory of session types is the *duality* relation. If the two endpoints of a communication channel have session types T and U, and each endpoint is used correctly according to its type, then the relationship $T \perp U$ guarantees that communication succeeds. Success means that when one endpoint

Types $\boxed{T, U, V, W}$

T ::=		Types:
	$\mathsf{end}_?$	end wait
	$\mathsf{end}_!$	end close
	$?T.T$	input
	$!T.T$	output

Figure 2.1 The syntax of types.

is waiting for a channel to be closed, the other endpoint closes the channel, and when one endpoint sends, the other endpoint receives, and vice versa; moreover, the value being sent has the type that the receiver expects.

Duality is defined inductively by the rules in Figure 2.2. Intuitively, the dual of output is input and the dual of input is output. In particular if U is dual to V, then $?T.U$ is dual to $!T.V$. The types $\mathsf{end}_!$ and $\mathsf{end}_?$ are dual to each other.

As an example, let us show that the type $!(?\mathsf{end}_!.\mathsf{end}_!).?\mathsf{end}_!.\mathsf{end}_!$ is dual to the type $?(?\mathsf{end}_!.\mathsf{end}_!).!\mathsf{end}_!.\mathsf{end}_?$.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{end}_! \perp \mathsf{end}_?}\ \text{D-E{\scriptsize ND}!}}{?\mathsf{end}_!.\mathsf{end}_! \perp !\mathsf{end}_!.\mathsf{end}_?}\ \text{D-?}}{!(?\mathsf{end}_!.\mathsf{end}_!).?\mathsf{end}_!.\mathsf{end}_! \perp ?(?\mathsf{end}_!.\mathsf{end}_!).!\mathsf{end}_!.\mathsf{end}_?}\ \text{D-!}$$

From the form of the rules it should be clear that duality is symmetric. We leave the proof as an exercise.

Exercise 2.1.1. Prove that the duality relation is symmetric but not reflexive or transitive.

We have presented duality as a relation, so the question arises: given a session type T, how many session types U are there such that $T \perp U$? It turns out that the answer is exactly one. This observation is captured by the next result.

Lemma 2.1.2 (Uniqueness of dual). *If $T \perp U$ and $T \perp V$ then $U = V$.*

Proof By rule induction on $T \perp U$. In case D-? we have $T = ?W.T'$ and $U = !W.U'$, with $T' \perp U'$. Because of the form of T, it must be the case that $T \perp V$ is also derived by D-?, so $V = !W.V'$ with $T' \perp V'$. By induction, $U' = V'$ and so $U = V$.

The case of D-! is symmetrical, and the cases of D-End? and D-End! are simpler. □

In Chapter 3 we introduce a notion of type equivalence, and then Lemma 2.1.2 is generalised so that the dual of a type is unique up to type equivalence.

Duality $\boxed{T \perp T}$

$$\mathsf{end}_? \perp \mathsf{end}_! \qquad \text{(D-END?)}$$

$$\mathsf{end}_! \perp \mathsf{end}_? \qquad \text{(D-END!)}$$

$$\frac{U \perp V}{?T.U \perp !T.V} \qquad \text{(D-?)}$$

$$\frac{U \perp V}{!T.U \perp ?T.V} \qquad \text{(D-!)}$$

Figure 2.2 Duality.

EXERCISE 2.1.3 (The dual of a session type). In the literature, the unique dual of a session type T is often written $T^\perp$. Give an inductive definition of the function $(\cdot)^\perp$, then show that $T \perp T^\perp$.

What is the dual of a dual? It's the original session type. The proof is left as an exercise. This result remains true in later systems, with a generalisation to type equivalence.

EXERCISE 2.1.4 (My dual's dual is myself). Prove that if $T \perp U$ and $U \perp V$ then $T = V$.

2.2 The Pi Calculus with Sessions

Session types describe structured communication among concurrent agents, so they fit naturally in a language with concurrency and communication. In order to focus on these core concepts, we use a form of pi calculus, featuring primitives for sending and receiving messages along communication channels, for parallel execution and for local scoping of channels. We begin with a subset of our final language, and extend it later in the chapter.

The syntax of our version of pi calculus builds on a countably infinite set $\mathbb{X}$ of variables. When writing definitions and examples we use lower case Roman letters to represent variables and the upper case Roman letters M, P and Q to represent processes.

Variables represent channel endpoints, which are used for communication. They are also used as placeholders for channel endpoints that are received during communication. We do not distinguish between these uses of variables. Many presentations of pi calculus use the term "name" where we use "variable". In Chapters 7 and 8 we introduce other kinds of values, for example data values and functions, and then we also use variables as placeholders for received values.

We define the syntax of processes by the grammar in Figure 2.3. The terminated process, or inaction, is denoted by $\mathbf{0}$. Channels are closed by $x?.P$ (*wait*) and $x!.P$ (*close*). The distinction between wait and close will become more significant in Chapter 9. The *receive* process $x?y.P$ receives, from the channel endpoint

Processes $\boxed{M, P, Q}$

P ::=		Processes:
	$\mathbf{0}$	inaction
	$x?.P$	wait
	$x!.P$	close
	$x?y.P$	receive
	$x!y.P$	send
	$P \mid P$	parallel composition
	$(\nu xy)P$	scope restriction

Figure 2.3 The syntax of processes.

represented by variable x, a variable that it uses to replace the variable y before continuing with the execution of process P. Conversely, the *send* process $x!y.P$ sends variable y on the channel endpoint represented by variable x and continues as P. The parallel composition $P \mid Q$ allows processes P and Q to proceed concurrently.

In interactive behaviour, variables come in pairs, called *co-variables*. The best way to understand co-variables is to think of them as representing the two endpoints of a communication channel. In order to communicate, threads do not need to share variables (we use the term *thread* for any process that is not a parallel composition). Since a channel is represented by a pair of co-variables, two threads may each hold one variable, allowing them to write or to read on the channel. When we introduce the type system, this mechanism will allow precise control of the resources needed for communication.

The syntax for scope restriction $(\nu xy)P$ combines two purposes. First, it simultaneously hides (or binds, or restricts the scope of) the variables x and y. Second, it establishes x and y as two co-variables, allowing communication to happen in process P, between a thread writing on x and another thread reading from y (or vice versa). We will see later, when we define a type system, that x and y must be different variables.

Now let us consider free and bound variables. A variable occurring in a process can be either *free* or *bound*. The bound variables are those that are used internally: y in $x?y.P$, and x and y in $(\nu xy)P$. All other variables are free and are available for external interaction by the process. The key point about bound variables is that their exact names don't matter and can be systematically changed without changing the meaning of a process. To make this more formal, we say that a *change of bound variable* in P is either (1) the replacement of a part $x?y.Q$ of P by $x?z.Q'$ where z does not occur in Q and Q' is obtained from Q by replacing all occurrences of y by z, or (2) the replacement of $(\nu xy)Q$ by $(\nu xz)Q'$ or of $(\nu yx)Q$ by $(\nu zx)Q'$, where again z does not occur in Q and Q' is obtained