Robust Communications in Erlang

A thesis submitted to

The University of Kent

in the subject of Computer Science

for the degree of

Doctor of Philosophy

By

Joseph Richard Harrison

January 2020

Abstract

Erlang is a dynamically-typed functional and concurrent programming language lauded by its proponents for its relatively simple syntax, process isolation, and fault tolerance. The functional aspect has rich features like pattern matching and tail-call optimisation, while the concurrent aspect uses isolated processes and asynchronous message passing to share state between system components. The two meet with pattern matching on mailboxes, which allows for a process to pick a message from its mailbox — potentially out of order — based on its structure, value, type, or a mixture thereof.

A strongly and dynamically typed language like Erlang can experience many kinds of runtime errors, such as ill-typed operands to arithmetic operators. The interaction between Erlang's type system and process mailboxes can lead to a more subtle runtime error which is harder to detect: orphan messages. As the types of messages are not checked either at compile time or runtime, a process can be sent a message which it will never receive. Essentially, non-trivial type discrepancies in Erlang programs can cause subtle bugs when communication is involved. These problems can be hard to detect and fix, with current solutions such as extensive testing and exhaustive model checking.

This thesis reports on work to detect communication-related type discrepancies in Erlang programs. A fragment of the Core Erlang intermediate format is modelled formally so that we can reason about the out-of-order communication in Erlang systems, particularly the dependencies between sent messages when determining whether orphan messages exist. Afterwards, a sub-typing relation based on Erlang's type system is introduced to clearly define the notion of an orphan message, forming the foundation of a system for automatic detection via a mix of static analysis and runtime verification. This culminates in automatic tooling to detect certain cases of communication discrepancies via static analysis, and automatic instrumentation of concurrent programs to detect and recover from more complicated cases at runtime.

Acknowledgements

My deepest appreciation goes to my supervisor throughout my studies, Simon Thompson. You have provided me with continuous support and feedback which has been vital to the successful completion of my thesis. I'd also like to extend my appreciation to Meng Wang, also a supervisor for part of my studies before leaving Kent for greener pastures. You made a fantastic supervision team that encouraged me whenever things seemed difficult. Your technical insight and academic rigour made me reflect on my work to make it better. The friendliness, understanding, and patience you have shown me will always be remembered.

Secondly, thanks to my examiners Olaf Chitil and Emilio Tuosto. Thank you for all of the time and effort that you put into reviewing my research. Your keen eyes and insightful contributions were greatly appreciated. Our discussions and your recommendations helped me to improve the structure, framing, and presentation of my work.

Furthermore, thank you to the remaining members of my supervision panel, Laura Bocchi and Sally Fincher. Laura kept a watchful eye on the overall direction of my research whilst Sally kept a watchful eye on me.

I am also grateful to Sonnary Dearden, the Support Coordinator for research students in our department. Throughout the years you have kept me on the right side of the University's seemingly infinite academic regulations and you have endured my endless procrastination.

Thanks to my parents, Sharon and Richard. You have helped me become the person I am today. Thank you for guiding me and letting me make my own decisions in life.

Finally, thank you to my partner, Jodie. I will always be grateful for the support you have given me. You have been here through the good days and the bad, always encouraging me and providing me your shoulder to lean on. Thank you for sticking by me even when I was irritable and demotivated. You were there whenever I needed you. You have been my rock.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

On 8th September 1998, the following message was posted to the `comp.lang.functional` newsgroup:

> Today, Ericsson releases its development environment Erlang/OTP, as Open Source. Open Source means that the source code is free to the public, and that anyone may use Erlang/OTP for building commercial applications without restrictions.
>
> Ericsson uses Erlang/OTP (Open Telecom Platform) to build carrier class products. For instance, Erlang/OTP has been used to develop the Ericsson AXD 301 ATM Switch, a system supporting mobility between different office sites with one personal DECT telephone, and a system providing 8 Mbits bandwidth to the home over twisted pair copper wires.
>
> [. . .]
>
> Erlang is a programming language which has many features more commonly associated with an operating system than with a programming language: concurrent processes, scheduling, memory management, distribution, networking, etc.
>
> Erlang/OTP makes it easier to build telecommunications products, with their high requirements for non-stop functionality, speed, distribution, and concurrency. Erlang/OTP supports a number of operating systems and processors, and can be integrated with different development languages.
>
> We want to spread the technology in order to speed development of Erlang/OTP, ensure a good supply of Erlang/OTP fluent programmers, minimize maintenance and development costs for the language, and keep the technology up to world class.

This message marked the beginning of over two decades of open source Erlang development. Today, Erlang is used in production by countless organisations: the widely used RabbitMQ message broker is written in Erlang (Pivotal Software 2019), as is the Riak distributed datastore (Klophaus 2010). The WhatsApp messaging service which sees over 1.5 billion monthly users also has backend components written in Erlang (Facebook Inc. 2018). Erlang also appears in parts of the internet's core infrastructure: Cisco estimates that 90% of internet traffic passes through an Erlang controlled node at one point or another (Bevemyr 2018).

To understand why these projects and organisations choose Erlang we should consider what Erlang/OTP has to offer. First, the language is oriented around its concurrency model: *lightweight isolated processes* which share memory by *message passing*. Each Erlang process has its own heap, stack, and *mailbox* which allows it to independently execute its own code and exchange messages asynchronously with other processes. This is coupled with the *de facto* virtual machine – the BEAM – which has been engineered with concurrency at its heart: processes and message passing are a core part of the BEAM, they are not merely userspace concepts. When these features are combined Erlang developers can create distributed, concurrent, and fault-tolerant applications with a touted "nine nines" uptime.

Like all other programming languages however, Erlang is not perfect. For starters, Erlang is dynamically typed: it is easy to accidentally transpose arguments in function calls, or pass non-numeric data to mathematical operators. Secondly, the language is garbage collected and the inter-process communication is *unbuffered*: messages are free to accumulate in a process' mailbox ad infintum until it either runs out of memory or is noticed by a monitoring system. To mitigate these perceived shortcomings, Erlang programmers typically rely on extensive test suites and profiling tools to ensure that their applications behave as expected. With this in mind, the Erlang/OTP suite has many tools to make the developer's job easier: testing libraries and utilities, memory and performance profiling tools, and the virtual machine is instrumented with real-time tracing tools. Furthermore a set of generic *behaviours* implement common process patterns to encourage code reuse: server processes, event handlers, and finite state machines to name a few.

Despite extensive testing and profiling however, it is easy for certain kinds of bugs to make it to production as testing and profiling are typically non-exhaustive. Furthermore, Erlang applications can run on multiple *nodes* distributed across a network. This opens the door to network issues where the messages sent between processes can get "lost", arrive after a significant delay (where the receiving process may have timed out while waiting), or accumulate in mailboxes without ever being processed. Experienced Erlang programmers often develop an intuition for the asynchronous communication model, but novices and students can often be left puzzled by communication related bugs in relatively simple programs.

This thesis aims to explore the kinds of *communication discrepancies* that can exist in Erlang programs: messages that are sent and never received and messages which are expected but which never arrive, for example. Starting with a formal model of Erlang we will explore how and where discrepancies can occur, and we will use a variety of static analysis and runtime verification techniques to automatically detect them. The aim of this work is to improve the robustness of communications in concurrent Erlang programs by analysing their source code at compile time to detect mismatches in the way processes communicate with each other, and to automatically instrument programs to protect from such mismatches at runtime.

Another aim of this thesis is compatibility with *existing* Erlang programs, such as the legacy code which has accumulated in over two decades of general availability of *Erlang/OTP*. To ensure compatibility with existing programs, therefore, the proposed approaches do not require Erlang programs to be written in a specific style, use specific libraries, or rely on manual code instrumentation. This approach is entirely *transparent* to the programmer: we are able to analyse code without modifying it or requiring any specific programming technique, and runtime verification can be achieved via automatic instrumentation in the form of compiler extensions and drop-in replacements for standard libraries. This means that programmers can benefit from the analyses presented herein using compiler flags and by substituting library dependencies instead of manually instrumenting or modifying their code.

A central aspect of the thesis is the novel *CoErl* language and its operational semantics which are introduced in chapter 4. While other formal models of Erlang exist, the choice to create a new language was made. There are several reasons for this:

1. We can closely follow the *Core Erlang 1.0.3 Language Specification* (Carlsson et al. 2004), which specifies the behaviour of the *Core Erlang* intermediate representation used in Erlang/OTP. This language has more consistent syntax, lexical scoping, and less complex operational semantics than the higher-level Erlang language.

2. We will add only the features necessary for reasoning about Erlang's concurrency: lightweight processes, asynchronous and unbounded message passing, and the pattern matching and guard evaluation rules which are used to interact with process mailboxes. We can therefore put aside the intricacies of Erlang's many data types and instead focus on a few representative types which showcase the behaviour of Erlang's *de facto* type system as a whole.

1.1 Thesis Structure

The rest of this thesis follows a linear structure, with each chapter building on the work of those prior:

- Chapter 2 - an introduction to Erlang's syntax, concurrency primitives, toolchain, and standard library.
- Chapter 4 - an operational semantics for a communicating fragment of the Core Erlang intermediate representation, based on the language's written specification.
- Chapter 5 - a trace based analysis of the operational semantics from chapter 4 in order to reason about how Erlang processes communicate and how discrepancies can occur.
- Chapter 6 - building a sub-typing system for Erlang which can be used to reason about the values that Erlang patterns, guards, and clauses will match, which can be used to reason about the types of messages that a process will receive when communicating.
- Chapter 7 - implementing a sub-typing algorithm for the type system from chapter 6 using *Binary Decision Diagrams* (BDDs).
- Chapter 8 - using the principles from chapter 5 and the type system from chapter 6 to statically analyse Erlang programs with the aim of detecting communication discrepancies, and automatically instrumenting programs to check the types of messages at runtime.
- Chapter 9 - we compare the work from the previous chapters to existing tooling and related research.
- Chapter 10 - an overall summary of the work, its relation to existing work, and potential avenues for future research.

1.1.1 Main Contributions

The main contributions of this thesis are:

- **An operational semantics for a communicating fragment of Core Erlang** based on the official language specification. The fragment features pattern matching, guard expressions, processes, and out-of-order asynchronous communication (chapter 4). This is accompanied by an analysis of communicating expressions using a labelled version of the operational semantics (chapter 5).

- **A Semantic Sub-Typing System for Erlang** with union, intersection, and negation types (chapter 6). This allows us to finely approximate the types of messages that will be received by a process based on the patterns, guards, and orders of clauses in receive expressions. A sub-typing algorithm based on *Binary Decision Diagrams* (BDDs) follows in chapter 7 which canonicalises types to compare them for semantic equality.
- **A lightweight hybrid analysis of Erlang communications** which uses type inference and the sub-typing algorithm to detect message passing errors (chapter 8). Static analysis is used to infer the types of send and received messages. Some communication discrepancies are automatically detected at compile time, and we also demonstrate how the same concepts can be used to protect processes at runtime.

1.1.2 Other Publications

Several contributions presented herein are also detailed in other published work, namely:

- 'Towards an Isabelle/HOL Formalisation of Core Erlang' (Harrison 2017) contains an earlier version of the CoErl language presented in chapter 4, namely its grammar, operational semantics, and several related theorems. In addition, foundational work for the trace analysis of CoErl contributed in chapter 5 is presented. This work also made use of the Isabelle/HOL interactive theorem prover to mechanically verify several theorems.
- 'Automatic Detection of Core Erlang Message Passing Errors' (Harrison 2018) presents a less developed version of the type system from chapter 6, with the notable omission of negation types. Portions of chapter 8 dedicated to static analysis (namely sections 8.2, 8.4.2 and 8.4.3) are also derived from this work.
- 'Runtime Type Safety for Erlang/OTP Behaviours' (Harrison 2019) demonstrates a lightweight runtime verification mechanism for Erlang programs based on an earlier version of the type system presented in chapter 6. This work serves as the foundation for chapter 8, specifically the portion concerning runtime verification (section 8.3).

Chapter 2

Background

The majority of the work in this thesis deals specifically with the Erlang programming language in one way or another. For example, code snippets throughout are written in Erlang, chapter 6 uses Erlang's datatypes and associated notation, and chapter 4 discusses the operational semantics of the language in detail. In addition, several contributions rely on an understanding of certain data structures and associated concepts, namely *Labelled Transition Systems* (LTSs) in chapter 5 and *Reduced Ordered Binary Decision Diagrams* (ROBBDs) in chapter 7. It is not intended that this chapter is read in isolation, but rather as necessary when reading other chapters.

This chapter is intended to serve as an overview for these topics: we start with an introduction to Erlang and its ecosystem in section 2.1, and then move on to discussions of *Labelled Transition Systems* (LTSs) and *Reduced Ordered Binary Decision Diagrams* (ROBBDs) (sections 2.2 and 2.3 respectively).

2.1 Erlang

Erlang is a language with two distinct aspects: functional programming and concurrent programming.

This section is meant to serve as an introduction to Erlang: it is *not* a complete Erlang tutorial or language reference. For those unfamiliar with Erlang, the book *Learn You Some Erlang for Great Good!: A Beginner's Guide* by Hebert is a great resource: it explores each aspect of Erlang's syntax in turn, and shows how fault tolerant *Open Telecom Platform* (OTP) applications can be built using Erlang's powerful runtime system and standard libraries (Hebert 2013). The official *Erlang/OTP Documentation* serves as a reference for the standard library, compiler, virtual machine, best design practices, and documentation for the rest of Erlang/OTP.

We first introduce the functional aspect, showing how we organise code into functions and modules, compile our code, and run it on the *Bogdan/Bjorn's Erlang Abstract Machine* (BEAM) virtual machine (section 2.1.1). The concurrent aspect follows, where we examine Erlang's concurrency primitives which form the basis of larger applications: lightweight processes and message passing (section 2.1.1). Afterwards, we look at the basic building block of real-world Erlang applications: generic behaviours which can be used to separate generic boilerplate code from application-specific code (section 2.1.3).

2.1.1 Functional Programming

With a syntax consisting of expressions and function declarations, Erlang is a functional programming language at heart. There are no looping constructs such as `for` and `while` and there are no global variables or mutable data. In fact, the only way to manipulate state in an Erlang system is by communicating with other processes, using an in-memory database provided by the runtime, or I/O. As a basic example, here is a definition of an Erlang function called `f` which adds together its two arguments, `X` and `Y`:

```
f(X, Y) -> X + Y.
```

The part on the left of the arrow (`->`) is called the *head* and consists of the function name and its arguments. On the right hand side is the *body*, containing one or more comma-separated expressions which comprise the definition of the function, where the value returned by the function is the result of evaluating the body's *last* expression. For example, this function doubles both of its arguments and then adds them together:

```
g(X, Y) ->
  XDoubled = X * 2,
  YDoubled = Y * 2,
  XDoubled + YDoubled.
```

Expressions are separated by commas, variables start with uppercase letters, and assignment appears to be performed using `=`.

In order to run these functions we must place them in a *module*, which is Erlang's chosen method for organising code. Each module starts with a name, then a list of the names of functions which will be exported (i.e. visible outside the module), followed by the function definitions themselves. If we do not export a function it will not be visible from outside the module, and it cannot be called from either the shell or from another module: the only way to access it will be

```erlang
-module(my_math).
-export([f/2,g/2]).

f(X, Y) -> X + Y.

g(X, Y) ->
  XDoubled = X * 2,
  YDoubled = Y * 2,
  XDoubled + YDoubled.
```

Listing 1: Erlang module `my_math.erl`

from *within* the module it's defined in. For our example we will put the functions `f` and `g` in the module `my_math`, which is defined in listing 1. The first line is the *module attribute* which specifies the name of our module (`my_math` in this case). On the second line is the export attribute which specifies that `f/2` and `g/2` should be visible outside the module. The syntax `f/2` means "the function called `f` with arity 2". Afterwards, we have the definitions of `f` and `g` as before.

This code should be saved in a file called `my_math.erl`, i.e. the name of the module followed by the extension `.erl`. To run the code in our module we must first compile it; this is because the BEAM virtual machine does not actually run Erlang code, but an imperative assembly-like bytecode. We compile Erlang code with the `erlc` executable included with Erlang/OTP:

```
joe@laptop:~ $ erlc my_math.erl
```

which – assuming no error message is printed – will create a file called `my_math.beam` in the same directory.

Now that we have compiled our code we can run it from an interactive shell, accessed via the `erl` executable. This starts the BEAM virtual machine, loads libraries, starts some background processes, and adds the current working directory to the module search path:

```
$ erl
Erlang/OTP 22 [erts-10.6] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]


Eshell V10.6  (abort with ^G)
1>
```

The syntax we use in the shell is the same as the syntax we use *inside* function definitions, for example:

```
1> 2 + 2.
4
```

```
2> X = 2.
2
3> Y = 4.
4
4> XDoubled = X * 2, YDoubled = Y * 2, XDoubled + YDoubled.
12
5>
```

To run the function g in our my_math module with the arguments 2 and 4 we use the name of the module and function separated by a comma, and then the arguments:

```
5> my_math:g(2, 4).
12
6>
```

Finally, we can exit the interactive shell using the special q function:

```
6> q().
ok
joe@laptop:~ $
```

These are the basics of programming in Erlang: writing function definitions in modules, listing them in the export attribute of the modules if desired, compiling them, and running them from the interactive shell. This gives us a foundation for exploring Erlang's more interesting functional programming features: pattern matching, guard expressions, and recursion with tail call optimisation.

Pattern Matching

Erlang programs typically make heavy use of *pattern matching*, where control flow decisions and variable assignments can be determined based on the value and structure of data using an expressive syntax. For example, the following function returns the value 'true' if its argument is the number 0 and 'false' otherwise:

```
is_zero(0) -> true;
is_zero(_N) -> false.
```

The function has two clauses instead of one, each separated by a semicolon. In the first clause we have used the value 0 instead of a variable name. At runtime, clauses will be tried in order: if the

```erlang
1  -module(my_tests).
2  -export([is_zero/1,is_zero_b/1]).
3
4  is_zero(0) -> true;
5  is_zero(_N) -> false.
6
7  is_zero_b(_N) -> false;
8  is_zero_b(0) -> true.
```

Listing 2: Erlang module `my_tests.erl`

function is called with the argument 0 the first clause will match and its body will be evaluated. If the first clause does not match, the second clause will be tried, and so forth.

Note that the order of the clauses *does matter*: clauses are tried in the order they appear, starting from the top. In listing 2 we have the module `my_tests` with two different versions of the `is_zero` function. When we compile this module we get a single warning which hints at what might happen when we run our functions:

```
$ erlc my_tests.erl
my_tests.erl:8: Warning: this clause cannot match because a previous clause at line
↪  7 always matches
```

When we run the function `is_zero` it behaves as expected:

```
1> my_tests:is_zero(1).
false
2> my_tests:is_zero(0).
true
3>
```

The function returns `'true'` only when called with the value 0. For the second function `is_zero_b` however, we can see that the first clause *always* matches, returning `'false'`:

```
3> my_tests:is_zero_b(1).
false
4> my_tests:is_zero_b(0).
false
5>
```

This happens because a variable in a pattern is (except in certain special cases) always *free*, i.e. it will match and then bind to any value. In fact, this is exactly what the = operator is doing: it is not actually *assignment*, but rather *matching*. So if we bind the value 2 to the variable X and then attempt to "reassign" it we will get an error:

```
1> X = 2.
2
2> X = 4.
** exception error: no match of right hand side value 4
3>
```

but we find that since X is bound to the value 2, it will pattern match against other instances of 2:

```
3> X.
2
4> X = 2.
2
```

Once a variable has been bound by pattern matching (such as with the initial `X = 2`) it can be used to match that value in later pattern matches. As an example of how this works the `my_shapes` module in listing 3 on the following page shows some functions for handling shapes represented as tuples, a built-in data type in Erlang. These examples are based on classwork from the CO545 undergraduate module at the University of Kent (Simon J Thompson and Orchard 2019).

A shape is either a rectangle of the form `{rectangle, Height, Width}` or a circle of the form `{circle, Radius}`. The braces indicate the start and end of a tuple with comma-delimited elements. In both cases the first element of each tuple is an *atom* which is a *datum* - a name equal only to itself commonly used in Erlang to distinguish between similarly structured data with different meaning.

The first two functions `is_rectangle` and `is_square` return `'true'` if the value passed to them is a rectangle or a circle respectively. These functions look at the structure and value of their argument: the first clause of `is_rectangle` matches only if the argument is a 3 element tuple whose first element is the atom `'rectangle'`, and the first clause of `is_circle` only matches if its argument is a 2 element tuple whose first element is the atom `'circle'`:

```
1> my_shapes:is_rectangle({rectangle, 3, 4}).
true
2> my_shapes:is_rectangle({circle, 9}).
false
```

```erlang
-module(my_shapes).
-export([is_rectangle/1,is_circle/1,is_square/1,
         perimeter/1,area/1]).

is_rectangle({rectangle, _, _}) -> true;
is_rectangle(_) -> false.

is_circle({circle, _}) -> true;
is_circle(_) -> false.

is_square({rectangle, X, X}) -> true;
is_square(_) -> false.

perimeter({rectangle, Height, Width}) -> (2 * Height) + (2 * Width);
perimeter({circle, Radius}) -> 2 * math:pi() * Radius.

area(Shape) ->
  case Shape of
    {rectangle, Height, Width} -> Height * Width;
    {circle, Radius} -> math:pi() * math:pow(Radius, 2)
  end.
```

Listing 3: Erlang module `my_shapes.erl`

```
3> my_shapes:is_circle({rectangle, 3, 4}).
false
4> my_shapes:is_circle({circle, 9}).
true
5> my_shapes:is_circle(something_else).
false
6>
```

Furthermore, the function `is_square` takes advantage of the fact that once a variable is bound in a pattern match, that variable can then be used to match against that value:

```
6> my_shapes:is_square({rectangle, 3, 4}).
false
7> my_shapes:is_square({rectangle, 3, 3}).
true
8>
```

The first clause of the function definition has the variable `X` *twice*. The variable `'X'` binds to the value of the second element because it is not currently bound to any value. Then it is matched against the third element, but because it is already bound to the value from the second element the pattern match will only succeed if the third element is the same as the second. We can see what is happening here by using the = operator:

```
8> {X, X} = {3, 3}.
{3,3}
9> {Y, Y} = {3, 4}.
** exception error: no match of right hand side value {3,4}
10> Y = 3.
3
11> Y = 4.
** exception error: no match of right hand side value 4
12>
```

Note that in some parts of these functions' heads we have used underscores. If a variable name is prefixed with an underscore the compiler will suppress any warnings about that variable not being used. Furthermore the variable name is _ (the underscore, also called the wildcard variable) is special because it is deliberately ignored by the compiler: the underscore never binds and can therefore be used to discard "unimportant" data like in the first three functions in listing 3, but it can also never be used to check for equality:

```
12> {_, _} = {2, 3}.
{2,3}
13> _.
* 1: variable '_' is unbound
14> {_Z, _Z} = {2, 3}.
** exception error: no match of right hand side value {2,3}
15>
```

We can also use a case expression to perform pattern matching on the result of an expression, as can be seen in the area function in listing 3. Regardless of whether function heads or case expressions are used for pattern matching, an error is encountered when no clause matches:

```
15> my_shapes:perimeter({triangle, 10, 20, 30}).
** exception error: no function clause matching
    my_shapes:perimeter({triangle,10,20,30})
     (my_shapes.erl, line 14)
```

```
16> my_shapes:area({triangle, 10, 20, 30}).
** exception error: no case clause matching {triangle,10,20,30}
     in function  my_shapes:area/1 (my_shapes.erl, line 18)
17>
```

Guard Expressions

Clauses can also have *guard expressions*: small Erlang expressions which return a boolean value `'true'` or `'false'`. The clause with the guard will only match when the pattern match is successful *and* the guard expression evaluates to `'true'`. We cannot use arbitrary expressions in guards, though: only operators and functions from a whitelist can be used (Erlang/OTP Team 2018, Expressions). This is because the Erlang compiler makes the assumption that guard expressions will terminate and that they will not have any side effects. The simplest way of achieving this was to restrict the syntax of guards to type checks and comparisons.

The module `my_numbers` in listing 4 on the next page shows some functions which use guard expressions for control flow. The function `sign/1` returns the sign of the number passed to it and returns the atom `'zero'` when passed 0 to avoid any arguments about whether it is positive:

```
1> my_numbers:sign(-4).
negative
2> my_numbers:sign(0).
zero
3>
```

The guard expressions appear after the keyword `when` in each function clause: the first returns true when `N < 0`, the second when `N` is numerically equal to 0 (i.e. it can be either a floating point number or an integer), and the third when `N > 0`. In Erlang the == operator means “equal to” and the =:= operator means “exactly equal to”:

```
3> 2 == 2.0.
true
4> 2 =:= 2.0.
false
5>
```

The second function `abs/1` returns the absolute value of a number by checking the result of `sign(N)`: if it is `'negative'` we subtract `N` from zero, and we return `N` in all other cases.

```erlang
1  -module(my_numbers).
2  -export([sign/1,abs/1,double/1,triple/1]).
3  
4  sign(N) when N < 0   -> negative;
5  sign(N) when N == 0 -> zero;
6  sign(N) when N > 0   -> positive.
7  
8  abs(N) ->
9    case sign(N) of
10     negative -> 0 - N;
11     _ -> N
12   end.
13 
14 double(N) -> N + N;
15 double(N) when is_float(N) -> N + N.
16 
17 triple(N) when is_integer(N) -> N + N + N;
18 triple(N) when is_float(N) -> N + N + N;
19 triple(N) when is_integer(N) or is_float(N) -> N + N + N.
```

Listing 4: Erlang module `my_numbers.erl`

Next, the `double/1` function doubles any number, but we have a superfluous second clause: the first clause will always match because the pattern `N` will always bind. The second clause uses the `is_float` *Built-In Function* (BIF)[1] which returns `'true'` if its argument is a floating point number. All floating point numbers will have already been handled by the previous clause because of its wildcard pattern, though. Luckily the Erlang compiler will catch this and print a warning, using a simple pattern exhaustiveness check:

```
$ erlc my_numbers.erl
my_numbers.erl:15: Warning: this clause cannot match
  because a previous clause at line 14 always matches
```

Unfortunately, this exhaustiveness check is quite simple: it doesn't have a comprehensive understanding of Erlang's type system or the relationship between these "type test" BIFs. For example, the compiler didn't generate a single warning about the `triple/3` function despite the fact that the third clause is entirely redundant: the first clause handles all integers, the second clause handles all floats, and the third handles integers and floats — but they have already been handled.

Guards are useful for distinguishing between different types of input and determining the range of a value, but excessive use of them can be considered bad practice in Erlang. Programmers

[1]Several of these *Built-In Functions* (BIFs) are whitelisted for use in guard expressions

are encouraged to not engage in "defensive programming" except at the boundaries of their application which deal with potentially malicious or malformed input data.

Recursion

Patterns and guards are the core features of Erlang typically used for functional programming: we can perform a complicated case analysis on function arguments and arbitrary expressions using a rich pattern and guard syntax.

One thing we have not yet addressed is looping constructs: iterating over some data structure such as a list, or continuing a computation until some condition is satisfied. The simple answer is that there is no syntax for writing an imperative loop exactly because Erlang is not imperative: there would be no way to mutate the current program state so that we could work towards the terminal case. Instead, Erlang programmers use recursion to write loops. We can call the same function again and again with different arguments until a base case is reached.

The functions in listing 5 on the following page show how recursion is used to build functions which loop. We can use `sum/1` to sum all of the integers between 0 and `N` inclusive:

```
1> my_rec:sum(10).
55
2>
```

and the `fib/1` function returns the Nth number of the Fibonacci sequence:

```
2> my_rec:fib(12).
144
3>
```

The rest of the functions in the `my_rec` module expect their inputs to be *lists*, one of Erlang's most frequently used data structures. A list in Erlang is either an empty (written `[]` and called *nil*) or a *cons cell* of the form `[H|T]`. In this case `H` is the *head* of the list and `T` is the *tail*. Lists in Erlang can be both *proper* and *improper*. A list is proper if it is *nil-terminated* (i.e. the tail of the last cons cell is `[]`) and it is improper if the last tail is anything else. Regardless, both of these types are lists are still considered to be lists by Erlang's `is_list` BIF:

```
1> is_list([3|[]]).
true
2> is_list([3|4]).
```

```erlang
-module(my_rec).
-export([sum/1,fib/1,sum_list/1,sum_list_better/1]).

sum(0) -> 0;
sum(N) -> N + sum(N-1).

fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

sum_list([]) -> 0;
sum_list([X|Xs]) -> X + sum_list(Xs).

sum_list_better(Xs) -> sum_list_acc(Xs, 0).

sum_list_acc([], Acc) -> Acc;
sum_list_acc([X|Xs], Acc) -> sum_list_acc(Xs, X + Acc).
```

Listing 5: Erlang module `my_rec.erl`

```
true
3>
```

For convenience, Erlang offers a comma-delimited notation for specifying proper lists:

```
3> [1,2,3,4] =:= [1|[2|[3|[4|[]]]]].
true
4>
```

With this in mind, we'll look at how the `sum_list/1` function behaves. It pattern matches on its only argument which it expects to be either an empty list or a cons cell. If the list is empty the function returns `0`. But if the list is a cons cell the head of the list is assigned to variable `X` and the tail of the list is assigned to `Xs` through a pattern match. Then, the head of the list is added to the sum of the tail of the list which is calculated by a recursive call to `sum_list`:

```
4> my_rec:sum_list([1,2,3,4]).
10
5> my_rec:sum_list([]).
0
6>
```

The `sum_list` function has a drawback, though: it is not *tail recursive*. In order to return a result for a cons cell it must hold `X` in memory while it calculates the sum of `Xs`, repeating the process until the end of the list is reached. The evaluation looks something like this:

```
sum_list([1,2,3,4])
= 1 + sum_list([2,3,4])
= 1 + 2 + sum_list([3,4])
= 1 + 2 + 3 + sum_list([4])
= 1 + 2 + 3 + 4 + sum_list([])
= 1 + 2 + 3 + 4 + 0
= 10
```

As Erlang is a strict language each expression must be evaluated to a value which in this case triggers repeated calls to `sum_list`. This requires the BEAM virtual machine to track of all of these "suspended" computations until all subexpressions have been evaluated.

By contrast, the `sum_list_acc` function is *tail recursive*: the final action the function performs is a call to itself. The `sum_list_better` function produces the same results as `sum_list`, but it does not require any suspended computations which we would need to return to later on:

```
sum_list_better([1,2,3,4])
= sum_list_acc([1,2,3,4], 0)
= sum_list_acc([2,3,4]), 1)
= sum_list_acc([3,4], 3)
= sum_list_acc([4], 6)
= sum_list_acc([], 10)
= 10
```

We can see that the expression never grows as we evaluate it, leading to less memory usage and – due to how the BEAM is designed – faster evaluation.

Now that we have looked at Erlang's functional aspect we can use what we have learned to build concurrent and fault-tolerant programs.

2.1.2 Concurrent Programming

Erlang's concurrency model is based on *lightweight processes* which communicate by *message passing*. A central tenet of this model is *strong isolation*: each process has its own memory and state and each of them is preemptively scheduled by the BEAM.

This isolation enables fault-tolerant application design because when one process crashes it doesn't cause any other processes to crash unless explicitly configured to do so. Developers can build trees of processes which isolate responsibility so that if one part of the system crashes it can be restarted independently of the rest of the system.

Processes

Computation in Erlang occurs in processes. To start one of these processes and perform some computation we call the `spawn/3` function from the standard library. Its arguments are the name of a module, the name of a function, and a list of arguments. So if we want to spawn a function which runs `my_numbers:sign` with the argument `4`, we do the following:

```
1> spawn(my_numbers, sign, [4]).
<0.80.0>
```

The value returned by `spawn` is the *Process Identifier* (PID) of the process created; each process in Erlang has a unique *Process Identifier* (PID) which can be used to address it.

Our process doesn't do anything useful at the moment: it is spawned, runs the `sign` function, and then exits. In order to do something useful, we will spawn a process which runs an anonymous function which calls `sign` then prints the result. We will use a variant of `spawn` which takes a closure as its argument:

```
2> F = fun() -> io:fwrite("Sign is ~p~n", [my_numbers:sign(4)]) end.
#Fun<erl_eval.21.126501267>
3> spawn(F).
<0.82.0>
Sign is positive
```

A process which ran the closure `F` was spawned, its PID was returned, and the process executed independently, ultimately printing "Sign is positive" to the shell.

Even the shell itself is a process, and when we spawn another process which encounters a runtime error we see that the shell continues as normal:

```
4> G = fun() -> io:fwrite("Result is ~p~n", [2 + hello]) end.
#Fun<erl_eval.21.126501267>
5> spawn(G).
<0.84.0>
=ERROR REPORT====
Error in process <0.84.0> with exit value:
{badarith,[{erlang,'+',[2,hello],[]},
           [...]]}
6>
```

The process with PID `<0.84.0>` crashed in the background while the shell continued to run.

Communication

Erlang processes communicate by *message passing*: each process has its own queue of incoming messages called a *mailbox*. A process can append a message to the mailbox of any process by *sending* a message, and a process can dequeue messages from its own mailbox by *receiving*. This process is completely explicit and entirely asynchronous: a sending process does not block, and a receiving process must explicitly access its mailbox to remove a waiting message.

Messages are usually sent with the `!` operator, which is an alias to `erlang:send/2`. The PID of the process we want to send a message to is placed before the `!`, and the message is placed after. For example, we can send the message `'hello'` to ourselves (as `self()` returns the PID of the process which called it):

```
1> self() ! hello.
hello
2>
```

Note that the printed value `'hello'` is *not* the message: the `!` operator's return value is the message that was sent.

We then receive messages using a `receive` expression which generally follows the same structure as a `case expression`: a sequence of clauses which each have a pattern and optional guard expression. To receive the message we just sent ourselves we can use a receive expression with a single clause containing a wildcard pattern:

```
2> receive
2>   X -> io:fwrite("Received: ~p~n", [X])
2> end.
Received: hello
ok
```

These `receive` expressions will attempt to find the first message in the mailbox which matches *any* of the clauses. If no matching message is found then the process waits until a new message arrives.

Expression	Mailbox state
-	[]
`self() ! hello`	[hello]
`self() ! 2`	[hello, 2]
`self() ! world`	[hello, 2, world]
`receive N when is_integer(N) -> expr end`	[hello, world]
`receive Y -> expr end`	[world]
`receive Z -> expr end`	[]

Table 2.1: Mailbox states for an out-of-order receive

The patterns and guards in the clauses of a `receive` expression can even be used to receive messages in a different order to which they were sent:

```
3> self() ! hello, self() ! 2, self() ! world.
world
4> receive N when is_integer(N) -> io:fwrite("Received: ~p~n", [N]) end.
Received: 2
5> receive Y -> io:fwrite("Received: ~p~n", [Y]) end.
Received: hello
6> receive Z -> io:fwrite("Received: ~p~n", [Z]) end.
Received: world
```

Table 2.1 shows what is happening here: we start with an empty mailbox, populate it with the messages `hello`, `2`, and `world`, then use pattern matching to dequeue them in a different order.

2.1.3 Writing Servers

A common task in Erlang is creating a process which acts as a server: it receives messages, performs computations, and responds to requests while maintaining some state. This is often achieved by writing a tight loop which uses recursion to update the server's state.

The code in listing 6a on the following page is an implementation of a small "counter" server. It maintains a number as its state, looping through receiving messages which either add or subtract from the current state, request a reply containing the current state, or instruct the server to stop. The `start` function spawns the server by passing the name of the current module (via the `?MODULE` macro), the name of the `loop` function, and a list of arguments to the `spawn` function. The `loop` function contains a single receive expression which pattern matches against messages in the mailbox, searching for the first message in the mailbox which matches a clause. If the message is a 2-tuple whose first element is the atom `'add'`, the server loops with the second

```erlang
-module(my_counter).
-export([start/1,loop/1]).

start(N) -> spawn(?MODULE, loop, [N]).

loop(N) ->
  receive
    {add, M} -> loop(N+M);
    {sub, M} -> loop(N-M);
    {get, From} ->
      From ! N,
      loop(N);
    stop ->
      io:fwrite("stopping.~n"),
      ok
  end.
```

(a) `my_counter` version 1

```erlang
-module(my_counter_v2).
-export([start/1,loop/1]).
-export([add/2,sub/2,get/1,stop/1]).

start(N) -> spawn(?MODULE, loop, [N]).

loop(N) ->
  receive
    {add, M} -> loop(N+M);
    {sub, M} -> loop(N-M);
    {get, From, Ref} ->
      From ! {res, N, Ref},
      loop(N);
    stop ->
      io:fwrite("stopping.~n"),
      ok
  end.

%% API functions

add(Pid, N) -> Pid ! {add, N}.

sub(Pid, N) -> Pid ! {sub, N}.

get(Pid) ->
  Ref = make_ref(),
  Pid ! {get, self(), Ref},
  receive
    {res, Res, Ref} -> Res
  end.

stop(Pid) -> Pid ! stop.
```

(b) `my_counter` version 2

Listing 6: Two different implementations of a counter server

element added to the current state, and similarly for `'sub'` in the second clause. When the server receives a 2-tuple with a first element of `'get'`, the server sends a message to the PID `From` containing the current server state, then looping again. The server continues looping like this until it receives the message `stop`, at which point it prints a diagnostic message and exits.

We'll start this server, modify its state several times, and then ask the server to send its state to us as a message:

```
1> Server = my_counter:start(0).
<0.80.0>
2> Server ! {add, 20}.
```

```
{add,20}
3> Server ! {sub, 5}.
{sub,5}
4> Server ! {get, self()}.
{get,<0.78.0>}
5> receive St -> io:fwrite("Received: ~p~n", [St]) end.
Received: 15
ok
```

While this works, it requires us to know the specific format of every message sent to and received from the server. An alternative is to write and expose an API which abstracts away all of the communication. The second version of the counter server in listing 6b does exactly this: we expose four new functions which can be used to interact with the server: `add`, `sub`, `get`, and `stop`. The server is mostly identical, with the exception of how the server responds to `get` requests: the client sends a *reference* in its request which the server includes in its response, which can be used to distinguish between similar messages in the mailbox. The function `make_ref` creates one of these globally unique references (Erlang/OTP Team 2018, Data Types):

```
6> make_ref().
#Ref<0.4171005631.3277848584.179206>
7> make_ref().
#Ref<0.4171005631.3277848584.179211>
```

Now we can interact with our server using the exported API:

```
8> Server2 = my_counter_v2:start(0).
<0.87.0>
9> my_counter_v2:add(Server2, 20).
{add,20}
10> my_counter_v2:sub(Server2, 5).
{sub,5}
11> my_counter_v2:get(Server2).
15
```

Generic Behaviours

The creators of Erlang/OTP noticed patterns in the way that processes in large applications was written: apart from specific business logic, many processes behaved identically. For example, some processes behaved like state machines: they would accept inputs in the form of messages, perhaps respond to the sender, and then transition to a new state. Others would behave like servers by implementing a tight loop: they would wait for messages, receive them and perform some action based on the content of the message, and then repeat this process until they were explicitly stopped.

These common patterns have been implemented as *generic behaviours* included with Erlang/OTP. To implement one of these behaviours two modules are required: the *generic* code (included in the standard library) and the *specific* code (written by the programmer). The generic code in the standard library is responsible for "running" the process: it sends and receives messages, contains error handling and logging code, and interacts with the virtual machine as necessary. The specific code is written by the programmer as part of their application in the form of an Erlang module containing *callback functions*.

The generic code is arranged in three modules which are part of the standard library (Erlang/OTP Team 2019d):

- **`gen_event`**: a generic event handler
- **`gen_server`**: a generic server process
- **`gen_statemgen_fsm`**: generic state machines.

To see how these behaviours can be used we have rewritten the `my_counter_v2` module from listing 6b using the `gen_server` library. Our implementation is shown in listing 7 on the next page. The first difference is the `behaviour` attribute on line 2, which instructs the Erlang compiler to check that we have exported all the callbacks required for the `gen_server` behaviour to work. Next, the `start` function now calls the `gen_server` library's start function; this will handle all process spawning and initialisation for us. This is followed by the same API functions as before, except they again call the `gen_server` library. We call `cast` when we want a fire-and-forget request where we don't expect a response, and we use `call` when we expect a response. Also, stopping a server is such a common task that there is a dedicated `stop` function available in the library.

We can interact with this server exactly as before, except that a few return values are different:

```
1> {ok, Server} = my_counter_gen:start(0).
{ok,<0.80.0>}
2> my_counter_gen:add(Server, 20).
ok
```

```
-module(my_counter_gen).
-behaviour(gen_server).

-export([start/1,add/2,sub/2,get/1,stop/1]).
-export([init/1,handle_call/3,handle_cast/2]).

start(N) -> gen_server:start(?MODULE, [N], []).

%% API functions

add(Server, N) -> gen_server:cast(Server, {add, N}).

sub(Server, N) -> gen_server:cast(Server, {sub, N}).

get(Server) -> gen_server:call(Server, get).

stop(Server) -> gen_server:stop(Server).

%% gen_server callbacks

init([N]) -> {ok, N}.

handle_cast({add, N}, State) -> {noreply, State+N};
handle_cast({sub, N}, State) -> {noreply, State-N}.

handle_call(get, _From, State) -> {reply, State, State}.
```

Listing 7: Counter server implemented using `gen_server`

```
3> my_counter_gen:sub(Server, 5).
ok
4> my_counter_gen:get(Server).
15
```

The callback functions for the `gen_server` behaviour come next and this is where the "specific" code for our server is written. First, the `init` function is responsible for setting up the server. For this example we just set the server's state to the argument passed to us by returning a tuple. The next function is `handle_cast` which is the callback function for handling requests initiated by a call to `cast`. The first argument is the content of the message and the second is the current server state. The expected return value is a tuple indicating how the server should respond and how its state should be updated. In both cases we specify that a reply should not be sent, and that the new server state is either added to or subtracted from. The `handle_call` function is similar, except it handles requests initiated by `call`. In this case our return value specifies that we should send a reply (`State` in this case) and that the new state of the server is the same as its original state.

When we call `gen_server:start` a new process is spawned containing all of the necessary machinery for receiving requests and sending responses, managing the state of the process, handling errors, and more. On the other side we have the `call` and `cast` functions which are responsible for constructing requests, sending them to the server, and waiting for responses.

These generic behaviours are complementary to Erlang's communication model and strong process isolation. Generic code is responsible for handling the process state and communication while the specific code handles all business logic. Furthermore, by writing and exporting an API for the server, the entire implementation can be abstracted away so that users of the module don't even have to consider *how* the module is implemented to use it in a concurrent application.

The behaviours also exploit Erlang's features: callback functions typically exploit pattern matching and guard expressions to separate clauses based on the "shape" of the request, and lightweight processes and asynchronous message passing are used to maintain state and isolate callers from unhandled errors and exceptions in the callees. This is the bedrock of most real-world Erlang applications: functional programming allows business logic to be implemented rapidly and relatively explicitly, while concurrent behaviours are used to maintain state and facilitate co-ordination between associated sub-systems.

2.2 Labelled Transition Systems

Chapter 5 uses transition systems to analyse the communicating behaviour of the CoErl language from chapter 4. Specifically, the analysis of CoErl relies on *labelled* transition systems, i.e. transition systems whose transitions are additionally equipped with labels. This section introduces the definitions necessary for chapter 5.

We start with the formal definition of a transition system, which is a set equipped with a binary relation over some subset of its elements:

Definition 2.2.1 (Transition System)**.** *A transition system is a pair of the form:*

$$(S, R)$$

where S is a set of states and $R \subseteq S \times S$ is a binary relation.

Furthermore, if two elements of S are related by R, then it is possible to make a *transition* from the first to the second:

Definition 2.2.2 (Transition)**.** *For any transition system (S, R), a transition is any pair $(x, y) \in (S \times S)$ such that $(x, y) \in R$, i.e. $R\ x\ y$.*

With this definition it is said that x *transitions to* y.

A related concept to the transition system is the *labelled* transition system, which is a transition system extended with a set of labels and the binary relation is replaced with a ternary relation:

Definition 2.2.3 (Labelled Transition System). *A labelled transition system is a triple of the form*

$$(S, A, R)$$

where S *is a set of states,* A *is a set of labels,* $R \subseteq S \times \alpha \times S$ *is a ternary relation.*

Labelled transition systems therefore have labelled transitions[2]:

Definition 2.2.4 (Labelled Transition). *For any labelled transition system* (S, A, R)*, a labelled transition is any triple* $(x, \alpha, y) \in (S \times A \times S)$ *such that* $(x, \alpha, y) \in R$*, i.e.* $R\ x\ \alpha\ y$.

In this definition, it is said that x *transitions to* y *via* (or *by*) α.

The small-step semantics cannot readily be modelled as an LTS as the semantics is neither equipped with a set of labels A, nor a ternary relation which labels steps in the system.

2.3 Reduced Ordered Binary Decision Diagrams

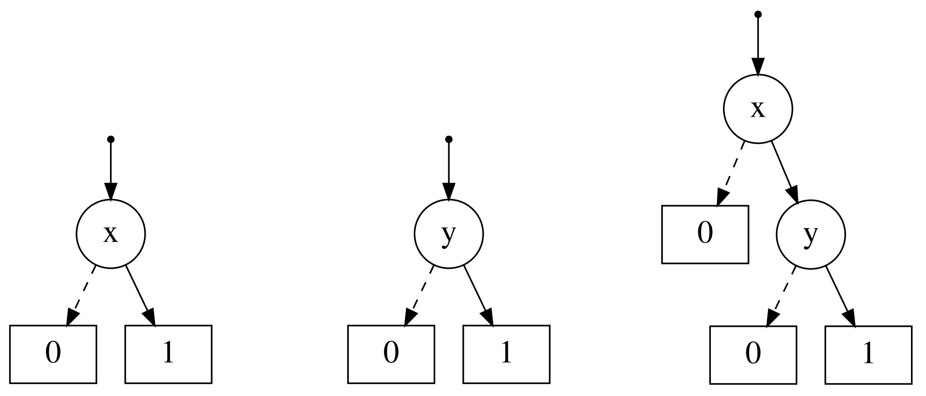
In this section we present well known definitions of BDDs, *Ordered Binary Decision Diagrams* (OBDDs), and ROBBDs and their associated construction algorithms (Huth and Ryan 2004) which will serve as the foundation of a specialisation BDD implementation for types.

A BDD is a directed acyclic graph used to represent boolean decision procedures. Figure 2.1 on the following page shows three BDDs: figure 2.1a represents the boolean formula x, figure 2.1b shows formula y, and figure 2.1c shows the formula $x \wedge y$. Each BDD consists of two types of node:

Definition 2.3.1 (BDD). *A Binary Decision Diagram is a directed acyclic graph with a single root. Each node in the graph is either:*

- $\langle x\ ?\ \mathtt{hi} : \mathtt{lo} \rangle$*: A "node" containing the name of a boolean variable with 2 labelled outgoing edges:* $\mathtt{hi}$ *and* $\mathtt{lo}$ (*drawn with a round border, where the* $\mathtt{hi}$ *edge is solid and the* $\mathtt{lo}$ *edge is dashed*)*; or*
- $\#\mathfrak{B}$*: A "leaf" containing a boolean value* **1** *or* **0** *with no outgoing edges* (*drawn with a square border*).

[2]though for brevity these are often just referred to as "transitions"



(a) BDD for formula x (b) BDD for formula y (c) BDD for formula $x \wedge y$

Figure 2.1: Naïve conjunction of BDDs

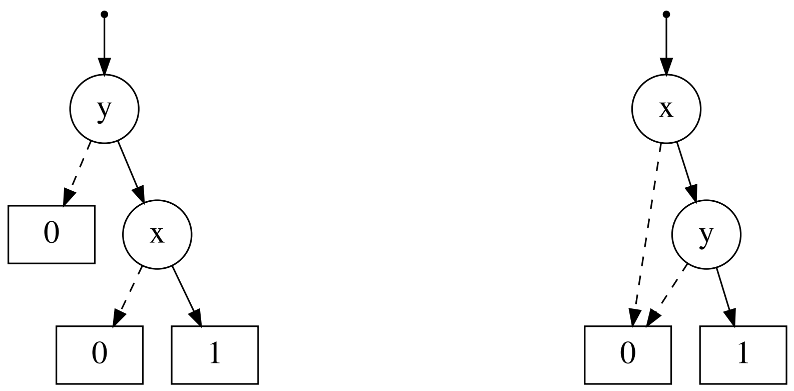
```
Data: A BDD b representing formula f with assignments ρ
Result: The result of the boolean formula f with assignments ρ
node ← b; while node is not a leaf do
    v ← variable(node)
    if value of v in ρ is 1 then
        node ← hi(v)
    end
    node ← lo(v)
    return value of v
end
```

Algorithm 1: BDD evaluation algorithm

Each BDD has a root node (indicated graphically using a solid arrow with a circular tail pointing into the top-mode node) and a leaf node is always reachable from any other node. To "evaluate" a BDD we need an assignment for each boolean variable which occurs in the graph and proceed as per the algorithm in algorithm 1. Starting at the root, we traverse the graph according to the assignments of variables: if a variable is assigned to **1** we proceed down the solid $\mathtt{hi}$ path, and if it is assigned to **0** we proceed down the dashed $\mathtt{lo}$ path. When we reach a leaf, we return the value contained within it.

For example, to evaluate the BDD in figure 2.1c using variable assignments $[x \mapsto 1, y \mapsto 0]$, we start at the root node x. As x is true, we follow the $\mathtt{hi}$ edge to reach node y. Checking the value of y, we note it is false, and we proceed down the $\mathtt{lo}$ edge to the leaf #**0**, and return false.

To check whether a given BDD is satisfiable, we can simply check for the existence of a path between the root node and a #**1** leaf: if no path exists then the formula is unsatisfiable, and if a path exists then it is satisfiable, and the combination of $\mathtt{hi}$ and $\mathtt{lo}$ edges along the path yields an assignment of variables which satisfies the formula. In figure figure 2.1c, the path between



(a) With redundant #**0** node and ordering $y < x$ (b) Without redundant #**0** node and ordering $x < y$

Figure 2.2: Different BDD representations of the formula $x \wedge y$

#**1** and the root goes via the $\mathtt{hi}$ edge of x (hence x must be **1**) and the $\mathtt{hi}$ edge of y, yielding the assignment $[x \mapsto 1, y \mapsto 1]$.

While figure 2.1c shows a single *possible* representation of the formula $x \wedge y$, there are other representations: the x and y nodes could be swapped (figure 2.2a), or one of the two #**0** leaves could be removed (figure 2.2b).

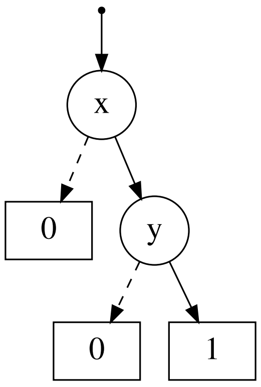
In figure 2.2a we have changed the *ordering* of variables, and in figure 2.2b we have *reduced* the BDD by removing redundant nodes. Both of these cases highlight a shortcoming of the rather lax definition of a BDD: the ordering and/or repetition of variables might yield different structures, as might different levels of "optimisation" to remove redundant nodes. Our BDDs are therefore *not* canonical: there can be several different representations of semantically equivalent formulas.

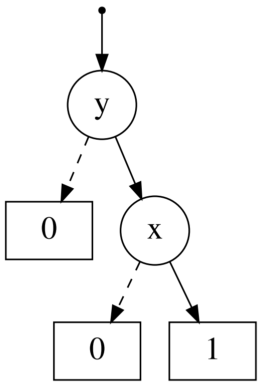
2.3.1 Ordering

The first step in making BDDs canonical is to introduce an *ordering* on the variables which occur in them. Specifically, we will restrict the order variables occur along any given path in the BDD, and also disallow more than one occurrence of each. The addition of an ordering creates an OBDD:

Definition 2.3.2 (OBDD). *An Ordered Binary Decision Diagram is a BDD with an ordering on variables $<$ such that for every path:*

- *variables occur in the order $<$; and*
- *no variable occurs more than once*


(a) with ordering $x < y$


(b) with ordering $y < x$

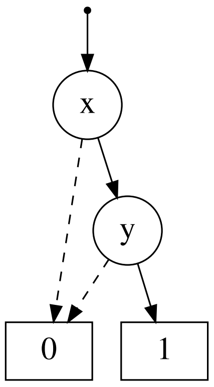

(c) with a duplicate **#0** leaf removed

Figure 2.3: Different OBDD representations of $x \wedge y$

To see how these rules affect the structure of BDDs, consider the different representations of $x \wedge y$ in figure 2.3. In figure 2.3a, the ordering $x < y$ is used, and in figure 2.3b the ordering $y < x$ is used. Both of these are valid OBDDs, but they use different orderings. As we will see later, our BDDs will be canonical *up to* the variable ordering, so BDDs with different orderings should not be compared for equality. Finally, the BDD in figure 2.3c is *not* a valid OBDD as it violates the second restriction: variables must not occur more than once in any path, but x occurs twice.

By adding an ordering on variables we transform BDDs into OBDDs, but there is still a possibility that two semantically equivalent graphs have different structures: redundant nodes.

2.3.2 Reduction

A common task for optimising compilers is to remove duplicated basic blocks. This is often achieved by removing all but a single copy of the code and replacing all references to the other copies with a reference to the single remaining block. With BDDs we can perform a similar optimisation: if there is more than one node with the same variable name, same `hi` edge, and same `lo` edge, we can remove all but one copy and redirect all incoming edges from the removed nodes.

Compilers also remove redundant tests - a common headache for those trying to write compiler benchmarks: if both branches of an if-then-else are identical, then the entire if-then-else can be replaced one of the two branches. Again, we can perform a similar operation on BDDs: if a node has a `hi` and `lo` edge with the same destination, the test is redundant, and can be removed.

This leads to the definition of a *reduced* OBDD:

Definition 2.3.3 (ROBBD)**.** *A Reduced Ordered Binary Decision Diagram is a OBDD where there are no:*

- *leaves with the same value; or*
- *nodes which have the same variable name,* `hi` *edge destination, and* `lo` *edge destination; or*
- *nodes where the* `hi` *and* `lo` *edges point to the same node.*

Figure 2.4 on the following page shows three different OBDDs for the formula $(x \vee \neg x) \wedge y$. The first BDD in figure 2.4a) contains a large amount of redundant information: there are duplicate leaves and there are two identical subgraphs. In figure 2.4b the duplicate leaves have been removed: only one copy of #**1** and #**0** remain. With this optimisation performed, we can now see that the two y nodes are identical, so we can remove one of them and redirect the incoming edges (figure 2.4c). Finally, observe that the x node is redundant: both the `hi` and `lo` edges have the same destination, so we remove it (figure 2.4d). This final OBDD satisfies all of the properties of a ROBBD: there are no duplicate leaves, duplicate nodes, or nodes with edges which point to the same location. The fact that we can remove x from the BDD entirely hints at the fact that variable x is redundant in the formula itself: $x \vee \neg x$ is a tautology.

The restrictions imposed on the structure of BDDs by the definitions of OBDDs and ROBBDs ultimately create a canonical form for any boolean formula *up to the variable ordering used* (Huth and Ryan 2004, ch. 6).

2.3.3 Combining ROBDDs: If-Then-Else

We can combine BDDs in various ways. For example, to negate a BDD we can swap all #**1** leaves with #**0** leaves, and vice versa. To take the conjunction of two BDDs we could replace all instances of #**1** in the first BDD with the root of (and the rest of) the second; similarly for disjunction, except replacing all instances of #**0**.

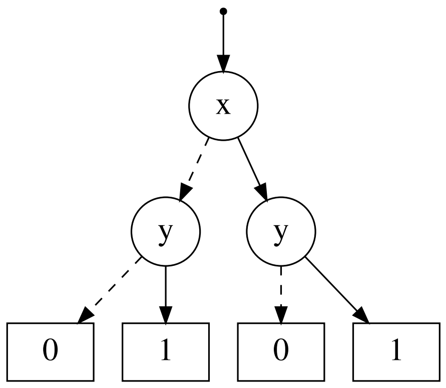
While this approach is workable, it does not yield *reduced* or *ordered* BDDs by construction: additional work is required to properly order variables and remove redundant nodes.

Another approach uses the knowledge that the $\wedge$, $\vee$, and $\neg$ operators in boolean logic can all be implemented using an if-then-else language construct:

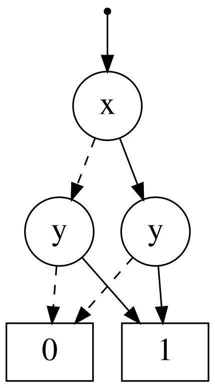
$$A \wedge B \iff \textbf{if } A \textbf{ then } B \textbf{ else } \mathbf{0}$$

$$A \vee B \iff \textbf{if } A \textbf{ then } \mathbf{1} \textbf{ else } B$$

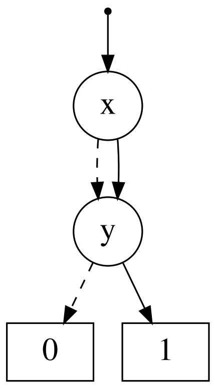
$$\neg A \iff \textbf{if } A \textbf{ then } \mathbf{0} \textbf{ else } \mathbf{1}$$



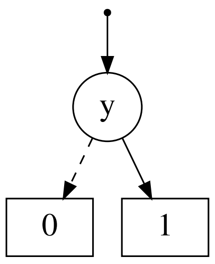
(a) with ordering $x < y$



(b) with duplicate leaves removed



(c) with duplicate y node removed



(d) with redundant x node removed

Figure 2.4: OBDD representations of $(x \vee \neg x) \wedge y$

Since each of these logical operators can be represented using an if-then-else construct, we only require one algorithm for constructing ROBBDs: a function which takes three ROBBDs (the test I, the true branch T, and the false branch E) and returns a ROBBDs equivalent to **if** I **then** T **else** E. This is not a novel approach, but it is significantly faster and more memory efficient than other construction techniques. The rest of this section presents algorithms originally seen elsewhere (Brace, Rudell and Bryant 1990).

The implementation relies on three key concepts:

1. Using a single *Multi-Rooted Directed Acyclic Graph* (MRDAG), where outgoing edges from nodes are pointers to other nodes, and where BDDs are represented by a pointer to a single node or leaf.

2. Shannon expansion, to restrict BDDs based on whether a variable is true or false.

3. A table of known nodes, to prevent the insertion of duplicate nodes into the MRDAG.

```
Function Find-Or-Create($v$, hi, lo, G)
    Data: A node $\langle v \text{ ? } \mathtt{hi} : \mathtt{lo} \rangle$ to add to the MRDAG G
    Data: The existing MRDAG G
    Result: A pointer to a node equivalent to $\langle v \text{ ? } \mathtt{hi} : \mathtt{lo} \rangle$
    if there is a node $n \in G$ such that $n = \langle v \text{ ? } \mathtt{hi} : \mathtt{lo} \rangle$ then
        return pointer to $n$
    end
    insert $\langle v \text{ ? } \mathtt{hi} : \mathtt{lo} \rangle$ into G
    return pointer to the inserted node
end
```

Algorithm 2: Find-Or-Create function for MRDAG based ROBDDs

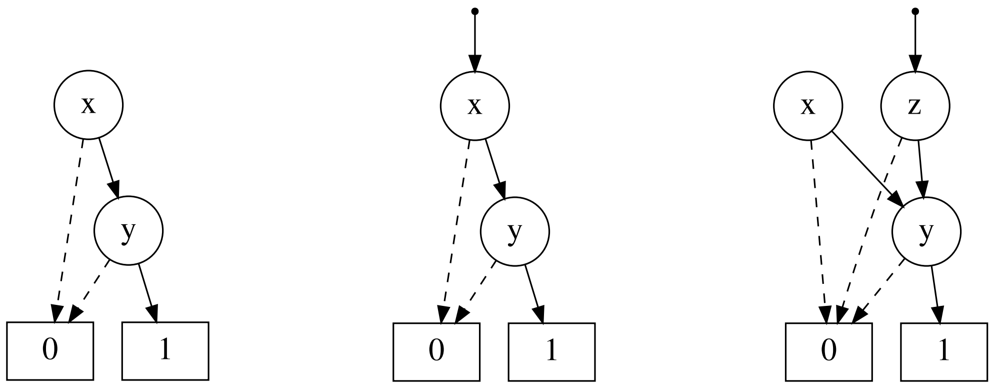
The first operation we define for these MRDAG based ROBBDs is the `Find-Or-Create` function which is responsible for inserting nodes into the *Directed Acyclic Graph* (DAG) (algorithm 2). Before inserting any node into the DAG, we check whether an equivalent node already exists, i.e. we search for another node with the same variable name and outgoing edges. The new node is only inserted if no equivalent node exists. If all insertions into the graph are performed with this function, we will never create a DAG where two identical nodes exist, which would violate one of the properties of ROBBDs.

In practice, the search for existing nodes is performed using a "unique" table which maps triples of node name, `hi` edge, and `lo` edge to pointers in the graph:

- When inserting a node to the graph, store a pointer to it in the hash table, using $(v, \mathtt{hi}, \mathtt{lo})$ as the key
- When searching for a node prior to insertion, look up $(v, \mathtt{hi}, \mathtt{lo})$ in the hash table. If a result is found, return the stored pointer, otherwise proceed to insert the node into the graph.

Figure 2.5 on the following page shows how the `Find-Or-Create` function operates; figure 2.5a is our initial MRDAG. In figure 2.5b we attempted to add the node $\langle x \text{ ? } \mathtt{ptr}(y) : \mathtt{ptr}(0) \rangle$ to the graph, but as an equivalent node already exists the graph is unmodified, keeping it reduced. In the case of figure 2.5c however, as no equivalent node $\langle z \text{ ? } \mathtt{ptr}(y) : \mathtt{ptr}(0) \rangle$ exists we insert the new node into the graph and return a pointer to it. However, if we use the variable ordering $x < y < z$ note that the `Find-Or-Create` function does not maintain the ordering of nodes required in ROBBDs: in figure 2.5c the z node occurs earlier than the x node. In this case the graph is still *reduced* but it is not *ordered*.

Preserving the order of nodes in the graph requires careful use of the `Find-Or-Create` function. To this end the rest of the approach - like many other algorithms for constructing ROBBDs - relies on Shannon expansion (Shannon 1949):



(a) initial BDD (b) inserting a duplicate node $\langle x \mathbin{?} y : 0 \rangle$ (c) inserting a unique node $\langle z \mathbin{?} y : 0 \rangle$

Figure 2.5: Examples of the `Find-Or-Create` function

Theorem 2.3.1 (Shannon expansion)**.** *For every boolean function* F*:*

$$F = (x \wedge F_x) \vee (x' \wedge F'_x)$$

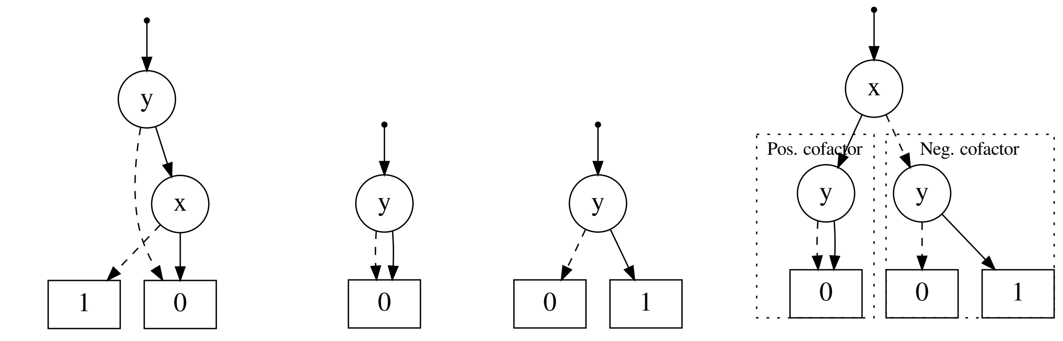
where x *is a boolean variable,* F_x *is the function* F *with* x *set to* **1***, and* F'_x *is the function* F *with* x *set to* **0***.*

F_x *and* F'_x *are called the positive and negative Shannon cofactors respectively.*

This identity allows us to "lift" any variable out of a boolean function and place it at the top level, which for BDDs means that we can lift any variable to the root of the graph. By strategically picking variables and repeatedly performing Shannon expansion on a BDD, nodes can be re-ordered so that they occur as per the $<$ relation.

In figure 2.6 on the next page we look at how this might be done. The BDD in figure 2.6a shows a ROBBD with variables ordered $y < x$. Assuming we want to change the ordering of variables to $x < y$, we would pick variable x to "factor out" of the BDD. Figure 2.6b shows the positive cofactor (where $x = 1$) obtained by redirecting the incoming edge of x to the destination of its hi edge and removing the x node, i.e. we set $\mathrm{hi}(x)$ to $\mathrm{hi}(x)$. Figure 2.6c shows the negative cofactor (where $x = \mathbf{0}$) by obtained by performing a similar operation on the lo edge of x: the incoming hi edge from y is redirected to the **1** leaf, i.e. we set $\mathrm{hi}(y)$ to $\mathrm{lo}(x)$. Finally, we combine the positive and negative cofactors in figure 2.6d by a case analysis on variable x: if x is true then we evaluate the positive cofactor, and if it is false we evaluate the negative cofactor. Although the BDD is no longer reduced (as we have more than one **0** leaf), we can see how shannon expansion can be used to re-order the variables.

Shannon expansion forms the basis of the if-then-else algorithm for constructing ROBBDs, shown in algorithm 3 on page 36. Assume that we want to combine three ROBBDs I, T, and E



(a) with ordering $y < x$ (b) pos. cofactor w.r.t x (c) neg. cofactor w.r.t x (d) with ordering $x < y$

Figure 2.6: Reordering nodes in a BDD via Shannon expansion

into a single ROBBD which represents **if** I **then** T **else** E. Also assume that all three BDDs are part of the same MRDAG, and that all of them use the same variable ordering. The first task is to deal with base cases:

- If I is a pointer to **1** we are attempting to represent **if 1 then** T **else** E which will always evaluate to T, so return T.
- If I is a pointer to **0** we are attempting to represent **if 0 then** T **else** E which will always evaluate to E, so return E.
- If T and E point to **1** and **0** we are attempting to represent **if** I **then 1 else 0** which is equivalent to I, so return I.
- If T and E are identical we are attempting to represent **if** I **then** T **else** E which will always yield the same result regardless of I, so return T or E.

The next step is to check whether we have called `If-Then-Else` with the same arguments before, and return the memoised result if we have. As BDD construction often leads to repeated function calls, memoisation often saves significant amounts of time.

The final case of the algorithm is where I, T, and E are actually combined using Shannon expansion. Assuming that I_x and I'_x are the positive and negative cofactors for I restricted on variable x, and likewise for T and E, then:

$$\textbf{if } I \textbf{ then } T \textbf{ else } E = \textbf{if } x = \textbf{1} \textbf{ then if } I_x \textbf{ then } T_x \textbf{ else } E_x \textbf{ else if } I'_x \textbf{ then } T'_x \textbf{ else } E'_x$$

In `If-Then-Else` this is done as follows:

1. Find the smallest variable x present in I, T, and E.

```
Data: An ordered and reduced MRDAG $G$
Function If-Then-Else($I, T, E$)
    Data: Pointers to nodes $I$, $T$, and $E$
    Result: Pointer to a node representing if $I$ then $T$ else $E$
    if $I$ points to #1 then
        return $T$
    else if $I$ points to #0 then
        return $E$
    else if $T$ points to #1 and $E$ points to #0 then
        return $I$;
    else if $T = E$ then
        return $T$
    else if If-Then-Else($I, T, E$) is memoised then
        $r \leftarrow$ memoised value for If-Then-Else($I, T, E$)
        return $r$
    else
        $x \leftarrow$ smallest variable in $I, T, E$ as per ordering $<$

        $(I_x, I'_x) \leftarrow$ Restrict($I, x$)
        $(T_x, T'_x) \leftarrow$ Restrict($T, x$)
        $(E_x, E'_x) \leftarrow$ Restrict($E, x$)

        $F \leftarrow$ If-Then-Else ($I_x, T_x, E_x$)
        $F' \leftarrow$ If-Then-Else ($I'_x, T'_x, E'_x$)

        $r \leftarrow$ Find-Or-Create($v, F, F'$)
        memoise If-Then-Else($I, T, E$) $= r$
        return $r$
end
```

Algorithm 3: If-Then-Else function for combining MRDAG based ROBDDs

2. Calculate the Shannon cofactors for each of I, T, and E relative to x, noting that the Restrict function will return pointers to existing nodes in the MRDAG, (i.e. ordering and reduction will be maintained when restricting).

3. Generate a pointer to the positive cofactor **if** I_x **then** T_x **else** E_x by calling If-Then-Else recursively.

4. Generate a pointer to the negative cofactor **if** I'_x **then** T'_x **else** E'_x by calling If-Then-Else recursively.

5. Find or create a new node in r the MRDAG for variable x, where the hi edge points to the positive cofactor and the lo edge points to the positive cofactor.

6. Memoise If-Then-Else$(I, T, E) = r$ and return r.

```
Function Restrict(v, p)
    Data: Name of a boolean variable $v$ to restrict on
    Data: Pointer $p$ representing an ROBBD in an MRDAG
    Result: Pointers to the positive and negative cofactors of $p$ restricted to $v$ respectively
    if p points to a node such that $\mathtt{variable}(p) = v$ then
        return $(\mathtt{hi}(p), \mathtt{lo}(p))$
    end
    else
        return $(p, p)$
    end
end
```

Algorithm 4: Restrict function for MRDAG based ROBDDs

Special care is taken to ensure the MRDAG remains ordered and reduced at all times. Firstly, the Shannon cofactors for I, T, and E are calculated using the `Restrict` function (algorithm 4). Consider that I, T, and E are all *ordered* BDDs which use the same ordering: the smallest variable in each will occur *at the root*. Therefore if $x = \mathtt{min}(I, T, E)$, for each of the three BDDs variable x either occurs at the root, or it does not occur at all (as it would violate the ordering if it appeared anywhere else). This means that the task of the `Restrict` function is simple: if the root is variable x, then the positive cofactor is the `hi` edge and the negative cofactor is the `lo` edge. If x is not present at the root however, then x does not occur in the BDD and the positive cofactors are the same: the original BDD.

The only other places that the graph can be modified (and the constraints violated) are in the recursive call to `If-Then-Else` (which is irrelevant if we always return an ordered and reduced MRDAG from the current call), and the call to `Find-Or-Create`. As we saw earlier `Find-Or-Create` will always keep the graph reduced, but it may create a node which violates the ordering of variables if given a variable which is larger than any of the variables present in the `hi` or `lo` sub-graphs. Fortunately, we call `Find-Or-Create` with variable x, which we already know is the smallest variable present, hence ordering and reduction will be maintained.

Assuming that hash table operations and graph insertion occur in constant time, then memoisation ensures that `If-Then-Else` has time complexity $\mathcal{O}(|I| \cdot |T| \cdot |E|)$, with a typical performance close to the size of the resulting boolean function (Brace, Rudell and Bryant 1990).

Chapter 3

Communication Discrepancies in Erlang

Despite all of this infrastructure – process isolation, asynchronous message passing, and generic server behaviours – Erlang programs can still crash or exhibit communications-related bugs.

Crashes tend to originate from the usual sources: I/O errors, unhandled corner cases, and malformed data. Erlang's dynamic type system doesn't make the situation any better: the lack of a static type system means that arguments of the wrong types are often passed to functions which cannot handle them properly. For example, we can cause our hand-written counter server to crash by attempting to add a non-numeric value:

```
1> Server = my_counter_v2:start(0).
<0.80.0>
2> my_counter_v2:add(Server, hello).
{add,hello}
=ERROR REPORT====
Error in process <0.80.0> with exit value:
{badarith,[{erlang,'+',[0,hello],[]},
           {my_counter_v2,loop,1,[{file,"my_counter_v2.erl"},{line,10}]}]}
```

and similarly with our gen_server version, except that the generic portion of the server means that we get a more detailed error report:

```
3> {ok, ServerGen} = my_counter_gen:start(0).
{ok,<0.81.0>}
4> my_counter_gen:add(ServerGen, hello).
ok
=ERROR REPORT====
** Generic server <0.81.0> terminating
** Last message in was {'$gen_cast',{add,hello}}
** When Server state == 0
** Reason for termination ==
** {badarith,[{erlang,'+',[0,hello],[]},
              {my_counter_gen,handle_cast,2,
                              [{file,"my_counter_gen.erl"},{line,23}]}
              [...]]}
=CRASH REPORT====
  crasher:
    initial call: my_counter_gen:init/1
    pid: <0.82.0>
    registered_name: []
    exception error: an error occurred when evaluating an arithmetic expression
      in operator  +/2
         called as 0 + hello
      in call from my_counter_gen:handle_cast/2 (my_counter_gen.erl, line 23)
      in call from gen_server:try_dispatch/4 (gen_server.erl, line 637)
      [...]
```

Note that in both cases the API function `add` returned a value which suggested the request was successful because sending a message is asynchronous, so the sender has no way of knowing the state of the server. When we make a `call` with our `gen_server` implementation things are more clearly wrong before the generic code in the library first checks whether the process exists before attempting to send a message:

```
1> {ok, ServerGen2} = my_counter_gen:start(0).
{ok,<0.80.0>}
2> my_counter_gen:stop(ServerGen2).
ok
3> my_counter_gen:get(ServerGen2).
** exception exit: {noproc,{gen_server,call,[<0.80.0>,get]}}
     in function  gen_server:call/2 (gen_server.erl, line 215)
```

This is just one example of a communication discrepancy: data flow analysis of the server's code would show that it expects a numeric value for addition, but any correctly structured tuple is

```erlang
-module(guarded_counter).
-export([start/1,loop/1]).
-export([add/2,sub/2,get/1,stop/1]).

start(N) -> spawn(?MODULE, loop, [N]).

loop(N) ->
  receive
    {add, M} when is_number(M) ->
      loop(N+M);
    {sub, M} when is_number(M) ->
      loop(N-M);
    {get, From, Ref} ->
      From ! {res, N, Ref},
      loop(N);
    stop ->
      io:fwrite("stopping.~n"),
      ok
  end.

%% API functions

add(Pid, N) -> Pid ! {add, N}.

sub(Pid, N) -> Pid ! {sub, N}.

get(Pid) ->
  Ref = make_ref(),
  Pid ! {get, self(), Ref},
  receive
    {res, Res, Ref} -> Res
  end.

stop(Pid) -> Pid ! stop.
```

(a) Hand-written server from listing 6b

```erlang
-module(guarded_counter_gen).
-behaviour(gen_server).

-export([start/1,stop/1,
         add/2,sub/2,get/1]).
-export([init/1,
         handle_call/3,handle_cast/2]).

start(N) ->
  gen_server:start(?MODULE, [N], []).

%% API functions

add(Server, N) ->
  gen_server:cast(Server, {add, N}).

sub(Server, N) ->
  gen_server:cast(Server, {sub, N}).

get(Server) ->
  gen_server:call(Server, get).

stop(Server) ->
  gen_server:stop(Server).

%% gen_server callbacks

init([N]) -> {ok, N}.

handle_cast({add, N}, State)
  when is_number(N) ->
    {noreply, State+N};
handle_cast({sub, N}, State)
  when is_number(N) ->
    {noreply, State-N}.

handle_call(get, _From, State) ->
  {reply, State, State}.
```

(b) gen_server version from listing 7

Listing 8: Counter servers with guards for numeric types

accepted. **There is a discrepancy between the type of message accepted by the receive clause and the type of value expected by functions which use it.**

Even if we added an `is_number` guard to the relevant clauses in the server code we would still see problems. The modules in listing 8 show the hand-written and `gen_server` based counter servers with guards for the `add` and `sub` cases.

When we run the gen_server version the server now crashes when we call add with a non-numeric value:

```
1> {ok, ServerGen} = guarded_counter_gen:start(0).
{ok,<0.80.0>}
2> guarded_counter_gen:add(ServerGen, hello).
=CRASH REPORT====
  crasher:
    initial call: guarded_counter_gen:init/1
    pid: <0.80.0>
    registered_name: []
    exception error: no function clause matching
    ↪  guarded_counter_gen:handle_cast({add,hello},0) (guarded_counter_gen.erl,
    ↪  line 23)
```

We could fix this with a catch-all clause on handle_cast but this is hiding the real communication discrepancy: **server processes written using the standard library's generic behaviours crash when they receive a request they are not programmed to handle**.

At first glance it looks like our other hand-written server might not crash when it's sent a non-numeric add message as the receive expression in the loop function will simply "skip over" these messages in the mailbox because they won't match any of the function clauses:

```
3> Server = guarded_counter:start(0).
<0.83.0>
4> % send a string instead of an integer
4> guarded_counter:add(Server, "42").
{add,"42"}
```

The server hasn't crashed because the message was never received by the add clause of the receive expression. But when we repeat this process many times we start to see another problem:

```
5> process_info(Server, [message_queue_len,total_heap_size]).
[{message_queue_len,1},{total_heap_size,233}]
6> integer_to_list(42). % converts an integer to a string
"42"
7> lists:foreach(fun(N) ->
7>   guarded_counter:add(Server, integer_to_list(N))
7> end, lists:seq(1, 10000)). % repeat 10000 times
ok
8> process_info(Server, [message_queue_len,total_heap_size]).
[{message_queue_len,10001},{total_heap_size,107801}]
```

As the messages are being placed on the process' heap in the BEAM the unreceived messages are consuming more and more memory: 107801 words instead of the earlier 233. This in turn will cause the BEAM to allocate more of its memory to the process, which is ultimately requesting more memory from the host operating system. This essentially constitutes a memory leak because the messages will never be received and will remain in the process' mailbox as long as it running, which may be days, weeks, or months in a real world application. Once the process exits however the BEAMs garbage collector will recover the memory. These are *orphan messages*: **messages can be sent to processes which will never receive them, leading to increased memory usage and decreased `receive` performance**.

These are just a few examples of the types of communication discrepancies that can occur in concurrent Erlang applications. Some of these discrepancies could be easily detected at compile time via static analysis. Others are readily apparent at runtime because they cause crashes, which means they could likely be detected through sufficient testing. Unfortunately, orphan messages are difficult to detect at compile time or runtime. The out-of-order receive behaviour combined with Erlang's expressive pattern and guard syntax makes it difficult to reason about all possible behaviours of an Erlang program during compilation. In addition, the memory leak at runtime is silent unless memory usage is being actively monitored, and other programming errors make it possible for programs continue to behave as designed even when orphan messages linger in mailboxes.

Chapter 4

CoErl: A Communicating Fragment of Core Erlang

Erlang is a mixture of two programming paradigms: functional and concurrent. The functional parts of Erlang are relatively unsurprising in the way they behave and are relatively well understood from a theoretical perspective: the language features pattern matching, higher-order functions, and data structures such as singly-linked lists and tuples. On the other hand, the concurrent aspect of the language presents interesting analytical challenges: Erlang allows a theoretically infinite number of self-contained processes – each with their own state – to communicate with each other through mailboxes, which allow messages to be received in a different order to which they are sent.

In order to understand how concurrent Erlang programs communicate, therefore, it will be useful to formalise the essential parts of the language with the goal of being able to analyse how each process in an Erlang system comes into existence, how processes send messages, and how these messages affect the behaviour of the processes which receive them.

Core Erlang will serve as the basis of the formalisation: it is an intermediate representation used in the Erlang/OTP compiler, in which all Erlang programs can be represented (Carlsson 2001). Core Erlang is chosen as the basis of the formalisation for several reasons:

- The Core Erlang Specification describes the syntax and operational semantics of the language in great deftail, although in textual form (Carlsson et al. 2004).
- Core Erlang features a reduced – but more explicit – syntax compared to Erlang
- The = (match) operator is eliminated in Core Erlang in favour of explicit variable bindings and pattern matching as appropriate

- Pattern matching only *binds* variables in Core Erlang: it never matches against already-bound variables.

Only a fragment of Core Erlang will be formalised, however: not all features of the language (and thus Erlang) are required to reason about processes and communication. On the other hand, enough of the language must be formalised in order to be useful enough to reason about real Erlang programs.

This chapter presents an operational semantics for a communicating fragment of Core Erlang, using the Core Erlang specification as a basis, but in two parts: a big-step semantics and a small-step semantics. These semantics will serve as a formal illustration of the behaviour of Core Erlang's communication model while also serving as a foundation for analysing the behaviour of communicating Erlang processes. Part of the semantics will be provided as function definitions: pattern matching, guard evaluation, and clause selection are all guaranteed to terminate, and are therefore easy to model formally. On the other hand, other aspects of the language are more difficult to model, such as non-termination: a core concept when writing long-lived concurrent Erlang programs. These aspects of the language will be modelled inductively, in a way which allows for partial evaluation of non-terminating programs. In the small-step semantics this will be achieved by the definition itself (where evaluation will be represented by a reflexive transitive closure over the semantics) and in the big-step semantics, this will be achieved by step indexing.

Overview This chapter begins with the syntax of a fragment of Core Erlang which is representative of the whole language, covering communication, recursion, pattern matching, and compound types (section 4.1). Afterwards, the function definitions which perform pattern matching, guard evaluation, and clause selection are given, accompanied with a full definition of the behaviour of receive expressions as a function over clause sequences and mailboxes (section 4.2). This is followed by definitions of the data structures used in the operational semantics: mailboxes, contexts, and processes (section 4.3).

We then give a big-step operational semantics for the communicating fragment, serving as an overview of how mailbox state propagates between sub-expressions (section 4.4). Small-step semantics follow, again defined inductively, modelling each distinct step in evaluation (section 4.5).

$$m ::= \texttt{module}\ l^a_m\ \texttt{where}\ l^a_f/n = \texttt{fun}(v_1, v_2, \ldots, v_n)\ \rightarrow\ e\ \texttt{end;}\ \ldots$$

$$\begin{aligned}
e ::=\ & l^\circ \mid v \mid [\,] \mid [\,e_h \mid e_t\,] \mid \{\,\} \mid \{e_1, e_2, \ldots, e_n\} \\
& \mid \texttt{case}\ \langle e \rangle\ \texttt{of}\ cs\ \texttt{end} \mid e_1\ \texttt{;;}\ e_2 \\
& \mid \texttt{let}\ v = e\ \texttt{in}\ e_b \mid \texttt{call}\ e_m : e_f\ (e_a) \\
& \mid \texttt{self} \mid e_p\ \texttt{!}\ e_m \mid \texttt{receive}\ cs\ \texttt{end}
\end{aligned}$$

$$\begin{aligned}
c &::= \langle p \rangle\ \texttt{when}\ g \rightarrow e \\
cs &::= c \mid c\ \texttt{;}\ cs
\end{aligned}$$

$$p ::= v \mid l^\circ \mid p = v \mid [\,] \mid [\,p_h \mid p_t\,] \mid \{\,\} \mid \{p_1, p_2, \ldots, p_n\}$$

$$v ::= \text{variable name}$$

$$g ::= \texttt{'true'} \mid \texttt{'false'} \mid \textbf{if}\ g\ \textbf{then}\ g\ \textbf{else}\ g \mid v\ \text{is}\ T$$

$$\begin{aligned}
T ::=\ & atom \mid boolean \mid float \mid integer \mid number \\
& \mid pid \mid port \mid reference \mid list \mid tuple
\end{aligned}$$

$$\begin{aligned}
l &::= l^\circ \mid [\,] \mid [\,l_h \mid l_t\,] \mid \{\,\} \mid \{l_1, l_2, \ldots, l_n\} \\
l^\circ &::= l^i \mid l^f \mid l^a \\
l^i &::= \text{integer literal} \\
l^f &::= \text{float literal} \\
l^a &::= \text{atom literal}
\end{aligned}$$

Figure 4.1: Syntax of CoErl

4.1 Syntax

The full syntax of the Core Erlang fragment is given in figure 4.1 on the previous page. The language consists of several distinct components: top-level module definitions (m), expressions (e), clauses used for pattern matching $(c$ and $cs)$, patterns (p), and literals (i.e. constant values, $l)$.

4.1.1 Modules (m)

All function definitions are arranged into *modules*. Each module has an atom as a name (l^a_m) and one or more function definitions of the form $l^a_f/n = \texttt{fun}(v_1, v_2, \ldots, v_n) \ \rightarrow\ e\ \texttt{end}$. Function definitions have a name l^a_f and an arity n (where $n \geqslant 0$), with each name/arity pair occurring at most once in each module definition. Each function has a head which declares the names of its arguments $(\texttt{fun}(v_1, v_2, \ldots, v_n))$ where n is equal to the declared arity of the function. Finally, each function has an expression e as a body.

4.1.2 Literals (l)

Some expressions can contain *literal values*, i.e. *constants*. These literals are defined by l and are either *atomic* or *compound*. The atomic literals are defined by l° and are either an arbitrary-precision integer (l^i), a floating point number (l^f), or an atom (l^a a datum, where each value of atom is equal only to itself) [1]. Integers and floats may be written using an optional sign, while atoms are delimited by a pair of single quotes, such as `'hello'`.

Alternatively, a literal may be a compound data type:

- $[\,]$: the empty list constructor, called *nil*.
- $[\ l_h \mid l_t\ ]$: a *cons* cell consisting of a head and a tail
- $\{\,\}$: an *empty tuple*.
- $\{l_1, l_2, \ldots, l_n\}$: a *tuple* consisting of any positive number of elements.

Together, these literals represent the most commonly-used data types in Erlang, with lists being constructed by nesting cons cells on the right-hand side and terminating them with a nil: in Erlang a programmer might write $[1, 2, 3]$ for a list of the integers 1 through 3, but the syntax in this fragment (and that of Core Erlang) requires that lists are constructed using the cons/nil notation of $[\,1 \mid [\,2 \mid [\,3 \mid [\,]\,]\,]\,]$.

[1] *atoms* should not be confused with *atomic literals*

Some examples of literals are `[]`, `[1 | 'hello' ]`, `[1 | [2 | [] ] ]`, and `{'req','save',...,`-2.1`}`. Also note that unlike some other languages, Erlang permits both well-formed (*proper*) and ill-formed (*improper*) lists: there is no requirement that last element in right-nested cons cells is nil.

4.1.3 Variable Names (v)

Variable names start with either an underscore or an upper-case letter. They may contain letters, numbers, underscores, and an `@` symbol. By convention, the Erlang/OTP compiler uses the `@` symbol for automatically-generated variable names, though this is not a requirement.

Variables starting with an underscore have a special meaning in Erlang: variables which are defined but never used generate a compile-time warning, but those which start with an underscore do not. Therefore, variables which are *intentionally* never used after being defined are prefixed with an underscore, which suppresses such compiler warnings. Unlike Erlang, however, this fragment of Core Erlang *will not* permit Erlang's behaviour where the special variable name `_` *never* binds, unlike all other variables:

```
1> {_, _} = {x,y}.
{x,y}
```

No variable names in this fragment will have special meaning, with the `@` and `_` symbols being used by solely by convention.

4.1.4 Patterns (p)

Patterns are an essential control flow mechanism for Erlang; they drive the clause selection process for functions, `case` expressions, and the receiving of messages.

The syntax of patterns is given in p and follows a similar structure to literals (l), albeit with a few additions. A pattern may either be any variable name (although variable names must not appear more than once in a pattern), an atomic literal, an alias of the form $p = v$, or one of the previously mentioned compound data types: nil, a cons cell, or a tuple. The alias $p = v$ assigns the value matched by p to variable v, where v may not occur in p.

The constructors for the compound data types have been replicated in p instead of using the definition of l to permit patterns to occur inside lists and tuples, while also eliminating any ambiguity as to whether a compound data type constructor in a pattern refers to the syntax of p or l.

4.1.5 Guards (g)

Guard expressions are used to further restrict the types of values accepted by clauses. A guard expression always evaluates to either `'true'` or `'false'`, where `'true'` represents success.

A guard is either the literal value `'true'` or `'false'`, an if-then-else expression allowing for boolean operations to be performed (e.g. conjunction, disjunction, and negation), and a type test of the form v is T.

4.1.6 Clauses (c)

Clauses build upon patterns and guards to form a control flow structure which appears in `case` and `receive` expressions. These always appear in semicolon delimited sequences of at least one clause (cs), and each clause is tried in order, starting with the first.

Each clause consists of a pattern p, a guard g, and a body expression e. Together, the pattern and guard form the *head* of the clause.

4.1.7 Expressions (e)

Expressions encapsulate all other definitions of the language. The simplest expressions consist of either atomic literals or variable names. This is followed by the compound types which are again placed here to remove ambiguity with the syntax of compound literals, and to allow arbitrary nesting of expressions inside the compound types.

The rest of the expression syntax is as follows:

- `case` $\langle e \rangle$ `of` cs `end` a *case* expression consisting of one *argument* expression e and a sequence of clauses cs.
- e_1 `;;` e_2: sequencing of two expressions e_1 and e_2.
- `let` $v = e$ `in` e_b: binding of variable v to e in e_b.
- `call` e_m `:` e_f $(e_1, e_2, \ldots, e_n)$: an *inter-module* function call, where e_m is the name of the module, e_f is the name of the function, and $(e_1, e_2, \ldots, e_n)$ are the arguments to use in the function call.
- `self`: an expression which evaluates to the unique process identifier for the process evaluating it.
- e_p `!` e_m: a concurrent expression, which sends message e_m to the process represented by e_p (processes are introduced in section 4.3.2 on page 58).

- `receive cs end`: receive a message from the mailbox according to the clause selection process (section 4.2).

4.2 Clause Selection Procedure

Clause selection uses pattern matching and guard evaluation to determine the next expression to evaluate. This appears in two forms in Core Erlang: either by evaluating the argument of a `case` expression before selecting a clause using the resulting value, or by iterating over the process' mailbox to pick a clause for a `receive` expression.

In both cases, this process consists of testing a value against each clause in a sequence. For a `case` expression, this process is performed once. In the case of a `receive` expression, this procedure is repeated for each message in the mailbox until a match is found (if there is one at all).

The first aspect is pattern matching: matching a value against a clause's pattern, which consists of various literal values, variable names, and compound data type constructors. Assuming that pattern matching succeeds, the clause's guard is then evaluated using any new variable bindings that were returned from pattern matching. If this succeeds, then the clause is selected; if not, the next clause is tried.

Each of these operations can fail, so a type must be used which encapsulates possible success or failure, similar to an option type:

Definition 4.2.1 (Failure Type)**.** *The failable type wraps a type* $\mathtt{a}$ *with two constructors: the* ***ok*** $\mathtt{a}$ *constructor represents success and contains a value of type* $\mathtt{a}$*, while* ***fail*** *represents failure and does not carry any value:*

$$\mathtt{Failable}\ \mathtt{a} \overset{\text{def}}{=} \boldsymbol{ok}\ \mathtt{a} \mid \boldsymbol{fail}$$

With this failable type, it is now possible to define the pattern matching, guard evaluation, and clause selection functions.

4.2.1 Pattern Matching

Pattern matching is the process of checking a value for some corresponding structure or values, optionally binding parts of the matched value to variables. For the purposes of pattern matching – and for the rest of the chapter – a value is equivalent to some literal defined by $\mathtt{l}^\circ$ in figure 4.1 on page 45. This is often used to deconstruct compound types such as lists and tuples, or to check for equality with some specific value (such as base cases in recursive functions), or to

$$\mathsf{pmatch}(p, \mathsf{val}) \stackrel{\text{def}}{=} \begin{cases} \textbf{ok}\ [\nu \mapsto \mathsf{val}] & \text{if } p = \nu \text{ where } \nu \text{ is a variable} \\ \textbf{ok}\ \mathsf{empty} & \text{if } p \in l^{\circ} \text{ and } p =:= \mathsf{val} \\ \mathsf{pmatch}(p_a, \mathsf{val}) \circ_f (\textbf{ok}\ [\nu \mapsto \mathsf{val}]) & \text{if } p = (p_a = \nu) \\ \textbf{ok}\ \mathsf{empty} & \text{if } p = [\,] \text{ and } \mathsf{val} = [\,] \\ \mathsf{pmatch}(p_h, \nu_h) \circ_f \mathsf{pmatch}(p_t, \nu_t) & \text{if } p = [\, p_h \mid p_t\,] \text{ and } \mathsf{val} = [\, \nu_h \mid \nu_t\,] \\ \textbf{ok}\ \mathsf{empty} & \text{if } p = \{\,\} \text{ and } \mathsf{val} = \{\,\} \\ \overset{n}{\underset{i=1}{\circ_f}}\ \mathsf{pmatch}(p_i, \nu_i) & \begin{array}{l}\text{if } p = \{p_1, p_2, \ldots, p_n\} \text{ and} \\ \mathsf{val} = \{\nu_1, \nu_2, \ldots, \nu_n\}\end{array} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

Figure 4.2: Pattern matching function $\mathsf{pmatch} : p \times \mathsf{val} \to \mathsf{Failable}\ \rho$

differentiate between data with different meanings to the programmer, such as tagging a tuple by setting its first element to a specific atom.

The variable binding portion of pattern matching will return a context which maps variable names to values:

Definition 4.2.2 (Contexts (ρ))**.** *A context is a partial mapping from variable names to values, denoted by the letter* ρ*. A variable is added to a mapping using the syntax* $\rho[x \mapsto v]$*, which binds the variable* x *to value* v *in context* ρ*, overwriting any previous value of* x*. Alternatively, the syntax* $\rho \circ \rho'$ *merges* ρ *and* ρ' *with any definitions in* ρ' *shadowing those in* ρ*. Values are looked up via the syntax* $\rho(x)$*, which returns the value of variable* x *in context* ρ*.*

The empty context is written empty.

For example, the following pattern matches a tuple with three elements, where the first element is the atom `'hello'`, with the other two elements assigned to the variables X and Y respectively:

```
{'add',X,Y}
```

The value $\{$`'add'`$, 2, 4\}$ would match the pattern, returning context ρ such that $\rho = [X \mapsto 2, Y \mapsto 4]$.

Figure 4.2 gives the definition of the pattern matching function pmatch, which takes as arguments a pattern p and value val, and returns a $\mathsf{Failable}\ \rho$, which may be either **ok** ρ or **fail**. As pattern matching does not always succeed, failure is captured using the type $\mathsf{Failable}$, which is defined in definition 4.2.1 on the previous page.

Definition 4.2.3 (Composition of $\mathsf{Failable}\ \rho$ ($\circ_f$))**.** *Two terms* x *and* y *of type* $\mathsf{Failable}\ \rho$ *may be composed such that if both represent success (via the* ***ok*** *constructor), then* $x \circ_f y = \textbf{ok}\ (x \circ y)$*:*

$$x \circ_f y \overset{\text{def}}{=} \begin{cases} \textbf{\textit{ok}}\ (x' \circ y') & \textit{if } x = \textbf{\textit{ok}}\ x' \textit{ and } y = \textbf{\textit{ok}}\ y' \\ \textbf{\textit{fail}} & \textit{otherwise} \end{cases}$$

The pmatch function operates by considering the syntax of the pattern p. In the case that the pattern is simply a variable ($p = v$), then the pattern match succeeds, binding variable v to val. For literals, pattern matching succeeds if val is equal to the literal l°, using Erlang's definition of equality (the `=:=` operator) and binds no variables. Pattern matching of aliases only succeeds when the sub-pattern p_a matches, which then finally binds v to val in the context returned from pattern matching p_a. When matching the empty list, val must also be an empty list constructor. For a cons cell, however, the function is recursive in the head and tail elements, requiring that val is a cons cell and that pattern matching succeeds for both the head and tail elements. Similarly to empty lists, the empty tuple checks that val is an empty tuple, again returning no bindings. The final successful case of pattern matching is a non-empty tuple: to succeed, both the pattern and value must have the same size (note that both tuples are of size n), and that pattern matching succeeds element-wise. The notation $\overset{n}{\underset{i=1}{\circ_f}}\ \mathit{pmatch}(p_i, v_i)$ is equivalent to $\mathit{pmatch}(p_1, v_1) \circ_f \mathit{pmatch}(p_2, v_2) \circ_f \ldots \circ_f \mathit{pmatch}(p_n, v_n)$. In all other cases, pattern matching fails, with pmatch returning the **fail** constructor.

The simplest successful pattern match binds a variable to a value by using a pattern which is a variable name, in this case X:

$$\mathit{pmatch}(X, 3.14) = \textbf{ok}\ [X \mapsto 3.14]$$
$$\mathit{pmatch}(X, \{\texttt{'add'}, 2, 4\}) = \textbf{ok}\ [X \mapsto \{\texttt{'add'}, 2, 4\}]$$

Regardless of the value used, pattern matching will always succeed for a pattern which is a variable name.

Aliases can be used to bind the "top level" value to a variable name. Without aliases, it is tedious to check for a certain data type constructor (such as a cons cell) and also bind the value to a variable, as the original value would have to be reconstructed using a cons cell and the variables bound during the pattern match. Instead, a pattern of the form $p = v$ allows binding of the value directly:

$$\begin{aligned} \mathit{pmatch}(\{\_X, \_Y\} = \mathrm{Pos}, \{2,4\}) &= \mathit{pmatch}(\{\_X, \_Y\}, \{2,4\}) \circ_f \textbf{ok}\ [\mathrm{Pos} \mapsto \{2,4\}] \\ &= \mathit{pmatch}(\_X, 2) \circ_f \mathit{pmatch}(\_Y, 4) \circ_f \textbf{ok}\ [\mathrm{Pos} \mapsto \{2,4\}] \\ &= \textbf{ok}\ [\_X \mapsto 2] \circ_f \textbf{ok}\ [\_Y \mapsto 4] \circ_f \textbf{ok}\ [\mathrm{Pos} \mapsto \{2,4\}] \\ &= \textbf{ok}\ [\_X \mapsto 2, \_Y \mapsto 4, \mathrm{Pos} \mapsto \{2,4\}] \end{aligned}$$

$$geval(g, \rho) \overset{\text{def}}{=} \begin{cases} true & \text{if } g = \texttt{'true'} \\ false & \text{if } g = \texttt{'false'} \\ \textbf{if } geval(i,\rho) \textbf{ then } geval(t,\rho) \textbf{ else } geval(e,\rho) & \text{if } g = \textbf{if } i \textbf{ then } t \textbf{ else } e \\ x \in T & \text{if } g = v \text{is T and } \rho(v) = x \\ x = l & \text{if } g = v =:= l \text{ and } \rho(v) = x \\ false & \text{otherwise} \end{cases}$$

Figure 4.3: Guard evaluation function $geval : g \times \rho \to \mathbb{B}$

In this case, each element of the tuple has been assigned a variable name which – by convention only – indicates we will not use the variable again, and the top-level tuple value has been bound to variable Pos.

Finally, when pattern matching compound values, it is necessary for the pattern to match the value element-wise, otherwise the entire pattern match fails:

$$\begin{aligned} pmatch(\{\texttt{'add'}, X, Y\}, \{\texttt{'sub'}, 2, 4\}) &= pmatch(\texttt{'add'}, \texttt{'sub'}) \circ_f pmatch(X, 2) \circ_f pmatch(Y, 4) \\ &= \textbf{fail} \circ_f \textbf{ok}\ [X \mapsto 2] \circ_f \textbf{ok}\ [Y \mapsto 4] \\ &= \textbf{fail} \end{aligned}$$

Although pattern matching succeeded for the second and third elements of the tuple, the first element failed as `'add'` is not equal to `'sub'` using Erlang's =:= operator.

In summary, the function pmatch implements all necessary pattern matching functionality for the fragment of Core Erlang given in this chapter. It is capable of deconstructing compound data types, checking for equality of literals, and aliasing. The function explicitly does not handle repeated variable names as they are not permitted in the syntax of Core Erlang patterns; this task is instead deferred to guard evaluation, which is discussed in the following section.

4.2.2 Guard Evaluation

The $geval$ function in figure 4.3 is used to evaluate guard expressions. Each of the `'true'`, `'false'`, and if-then-else cases are relatively straightforward. For type checks of the form v is T the function checks whether the value of variable v in the environment ρ is a member of the type T. Finally, if the guard is an equality check on a literal ($v =:= l$) the function looks up the value of variable v and checks for syntactic equality.

Building on the Core Erlang specification, we assume that there are no free variables in the guard, i.e. the environment ρ contains all the necessary variables:

$$\mathit{cmatch}(\langle p \rangle \texttt{ when } g \to e, \mathit{val}, \rho) \overset{\text{def}}{=} \begin{cases} \textbf{ok } \rho \circ \rho' & \text{if } \mathit{pmatch}(p, \mathit{val}) = \textbf{ok } \rho' \\ & \text{and } \mathit{geval}(g, \rho \circ \rho') = \mathit{true} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

Figure 4.4: Clause matching function $\mathit{cmatch} : c \times \mathit{val} \times \rho \to \mathit{Failable}\ \rho$

$$\mathit{cselect}(cs, \mathit{val}, \rho) \overset{\text{def}}{=} \begin{cases} \textbf{ok } (\mathit{expr}(c), \rho') & \text{if } cs = c; cs' \text{ and } \mathit{cmatch}(c, \mathit{val}, \rho) = \textbf{ok } \rho' \\ \mathit{cselect}(cs', \mathit{val}, \rho) & \text{if } cs = c; cs' \text{ and } \mathit{cmatch}(c, \mathit{val}, \rho) = \textbf{fail} \\ \textbf{ok } (\mathit{expr}(c), \rho') & \text{if } cs = c \text{ and } \mathit{cmatch}(c, \mathit{val}, \rho) = \textbf{ok } \rho' \\ \textbf{fail} & \text{otherwise} \end{cases}$$

Figure 4.5: Clause selection function $\mathit{cselect} : cs \times \mathit{val} \times \rho \to \mathit{Failable}\ (e \times \rho)$

Assumption 4.2.1 (Domain of geval)**.** *For all contexts* ρ *passed to* geval*, it is assumed that* $\mathit{vars}(g) \subseteq \mathit{dom}(\rho)$*. Note that* ρ *need not be the smallest* ρ *such that* $\mathit{dom}(\rho) \subseteq\in \mathit{vars}(g)$*.*

4.2.3 Clause Selection

The clause selection procedure combines pattern matching and guard evaluation from the previous sections. Clause selection proceeds by testing each clause in order, starting with the first: a pattern match is attempted against the value provided. If the pattern match succeeds, then the guard is evaluated in a modified context which contains any new bindings from the pattern match. If this guard evaluation succeeds, then the clause is selected, with evaluation continuing with the body of the clause and the updated context. In the case that either the pattern match or guard evaluation fails, then the next clause is tried, and so forth, until a clause which matches is found. When no clause in the given sequence matches, the clause selection fails.

This selection procedure is modelled in two parts: the task of matching a given value against the pattern and guard of a *single* clause, and separately the task of repeating this process for each clause in a sequence.

The cmatch function in figure 4.4 matches against a single clause. It takes as inputs a clause of the form $\langle p \rangle \texttt{ when } g \to e$, a value val, and a context ρ which contains any already-bound variables. First, pattern matching against the clause's pattern is attempted. If this succeeds, the guard is evaluated in the context of $\rho \circ \rho'$, which merges the original context ρ with any new variables bound in the pattern p. No shadowing of variables may occur, however:

Assumption 4.2.2 (Domain of ρ in $\mathtt{cmatch}$)**.** *For all contexts ρ provided to $\mathtt{cmatch}$, there must not be any already-bound variable ν in ρ such that ν appears in the clause's pattern p:*

$$\mathrm{dom}(\rho) \cap \mathrm{vars}(p) = \emptyset$$

Instead, all equality checks must be performed in the guard of the clause using the =:= operator.

Once $\mathtt{cmatch}$ succeeds, the clause is selected; when $\mathtt{cmatch}$ fails, the next clause is attempted. This process can fail in the case that no clause matches the value and context. The $\mathtt{cselect}$ function in figure 4.5 on the preceding page iterates over a clause sequence exactly as just described: either the first clause matches (and the corresponding clause body and updated context are returned), or the next clause is attempted until either a match is found or there are no more clauses to be tried. This function takes as arguments a sequence of clauses cs, a value val, and a context ρ and returns either an **ok** $(e \times \rho)$ or **fail** which represent success and failure respectively.

4.3 Processes, Mailboxes, and State

With the clause selection procedure defined, it is now possible to reason about and define *processes* and *mailboxes* which lie at the heart of Erlang's concurrency model.

A mailbox is an unbounded queue of messages sent to a process, where they are ordered as they arrive, such that the message which arrived most recently is at the "back" or "end" of the queue, and the oldest message is at the "front" or "start". Each of these mailboxes is associated with a process, which evaluates an expression in isolation from other processes except for communication: the sending and receiving of messages. All processes have their own unique identifier called a PID which allows processes to communicate with each other by sending messages to specific PIDs.

4.3.1 Mailboxes

Each Erlang process has a mailbox, which can be represented as a singly-linked list of values. Figure 4.6 on the next page shows a visual representation of a mailbox: the first message m_1 is at the front of the mailbox (in position 1), while m_2 is in the second position, and so forth.

The size of a mailbox associated with a process is unbounded: the operational semantics of Core Erlang impose no limit on the size of a mailbox, and in practice the size of mailboxes is limited only by available memory.

In Core Erlang, mailboxes are not interacted with directly and are not accessible as any kind of in-language data structure. Instead, two different operations are performed on mailboxes to

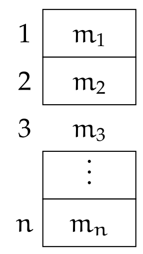


Figure 4.6: A mailbox of size n.

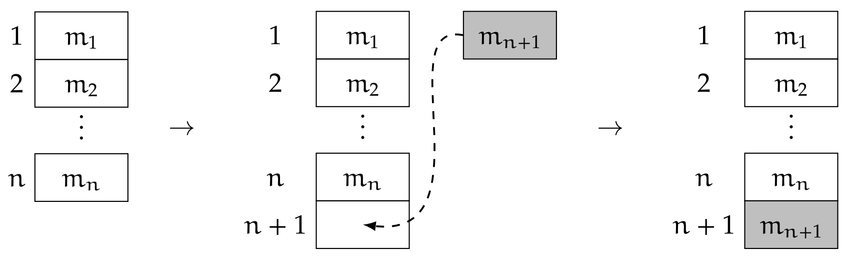


Figure 4.7: Message m_{n+1} arriving in a mailbox which already contains n messages

manipulate them: *arrival*, where new messages are placed at the end of the mailbox, and *receive*, where pattern matching and evaluation of guard expressions is used to remove messages from the mailbox, potentially out-of-order.

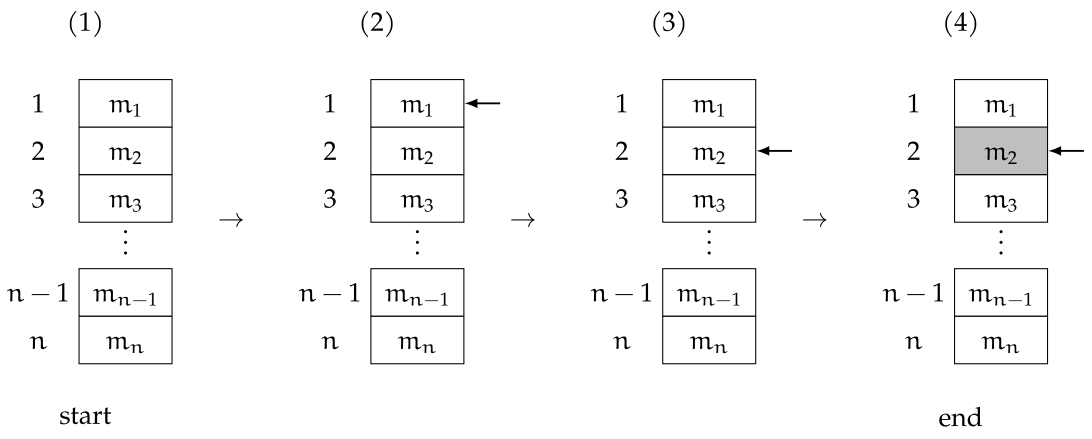
Arrival

When a message is sent to a process, perhaps by another process, or as a result of some external source (the *Erlang Runtime System* (ERTS), network sockets, etc.), the message will be placed at the end of the destination process' mailbox. In practice, the concurrent nature of the BEAM makes this operation extremely complicated: locks, off-heap allocation, and multiple threads are used to avoid deadlock situations. For the purposes of exploring the semantics of Core Erlang, however, it is sufficient to model this operation as a list append operation.

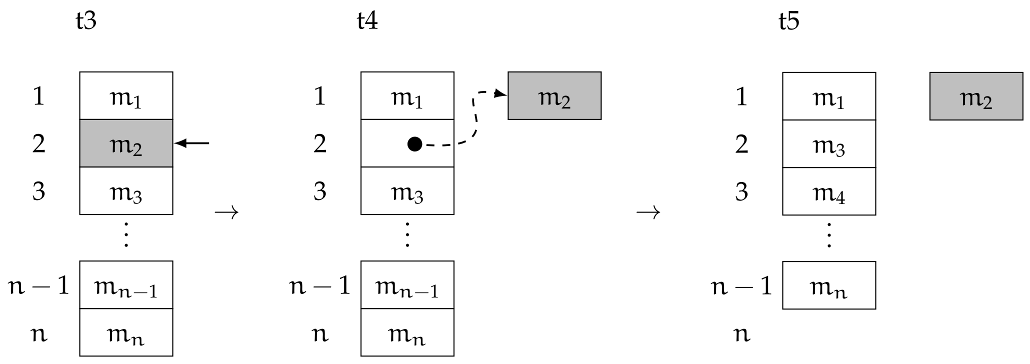
Figure 4.7 illustrates the arrival of messages in a mailbox: assuming that a mailbox already contains n messages (left), the arrival of the message m_{n+1} will cause the size of the mailbox to grow by 1 (centre), and the message is places at the *end* of the mailbox, in position $n + 1$ (right).

Receive Operation

The dual to arrival is *receiving*: removing a message from a mailbox via clause selection (section 4.2 on page 49). Receiving a message takes place in two stages: *selecting* a message and



(a) Selecting message m_2 from a mailbox



(b) Removing message m_2 from a mailbox

Figure 4.8: Receiving message m_2 from a mailbox

$$\mathsf{mbselect}(cs, mb, \rho) \overset{\text{def}}{=} \begin{cases} \textbf{ok}\ ((c, \rho), m) & \text{if } \mathsf{cselect}(cs, m, \rho) \ \neq\ \textbf{ok}\ (c, \rho) \\ & \text{and } mb = [\, m \mid mb' \,] \\ \mathsf{mbselect}(cs, mb', \rho) & \text{if } \mathsf{cselect}(cs, m, \rho) \ =\ \textbf{fail} \\ & \text{and } mb = [\, m \mid mb' \,] \\ \textbf{fail} & \text{if } mb = [\,] \end{cases}$$

Figure 4.9: Mailbox selection function $\mathsf{mbselect}$

simultaneously selecting a clause, and *removing* a selected message from a mailbox.

The selection function $\mathsf{mbselect}$ is responsible for selecting a message from a mailbox mb using some clauses cs in context ρ. For a given cs, mb, and ρ, $\mathsf{mbselect}$ returns either the first message to match the given clauses cs, or it fails. Each message in the mailbox is tried in order, starting with the first message: if clause selection succeeds, the result of selection and the first message is returned. If the first message does not match, successive messages in the mailbox are

$$mbremove(m, mb) \overset{\text{def}}{=} \begin{cases} mb' & \text{if } mb = [\, m' \mid mb' \,] \text{ and } m = m' \\ [\, m' \mid mbremove(m, mb') \,] & \text{if } mb = [\, m' \mid mb' \,] \text{ and } m \neq m' \end{cases}$$

Figure 4.10: Mailbox removal function $mbremove$

$$mbreceive(cs, mb, \rho) \overset{\text{def}}{=} \begin{cases} ((e, \rho'), mbremove(m, mb)) & \text{if } mbselect(cs, mb, \rho) = ((e, \rho'), m) \\ \textbf{fail} & \text{otherwise} \end{cases}$$

Figure 4.11: Mailbox receive function $mbreceive$

tried until either selection succeeds, or no messages remain, in which case $fail$ is returned. Note that it is a common case for selection to $fail$, as processes often await messages which have not yet arrived.

As an example of this mailbox selection process, assume a mailbox mb such that some messages m_1 and m_2 are at its head, such that $mb = [m_1, m_2, \ldots]$. Furthermore, assume that for some cs and ρ, $cselect(cs, m_1, \rho) = fail$ and $cselect(cs, m_2, \rho) \neq fail$. With these assumptions, mailbox selection proceeds as shown in figure 4.8a on the preceding page:

1. The mailbox mb contains n messages, with m_1 and m_2 at the head.
2. Clause selection is attempted with m_1, which fails: $cselect(cs, m_1, \rho) = fail$.
3. The next message is chosen (m_2) and clause selection is attempted, which succeeds: $cselect(cs, m_2, \rho) \neq fail$.
4. The message m_2 is selected, and the result $(cselect(cs, m_2, \rho), m_2)$ is returned.

Once a message has been selected, such that $mbselect(cs, mb, \rho) = ((c, \rho'), m)$, the message m must be removed from the mailbox. This is performed by $mbremove$ (figure 4.10) which takes m and mb as arguments, and returns mb' which is mb with the first message m' (such that $m = m'$) removed. Figure 4.8b on the preceding page shows the mailbox removal function operating on m_2 (which was selected in figure 4.8a on the previous page).

Finally, the functions $mbselect$ and $mbremove$ are combined into a single operation called $mbreceive$ which both performs clause selection and removes messages from the mailbox (figure 4.11). In the case that a message is selected from the mailbox, $mbreceive$ returns the result of clause selection *and* the original mailbox with the first matched message removed.

Again, failure is indicated by returning $\mathtt{fail}$, and is also a common occurrence in real-world Erlang programs.

The evaluation of the $\mathit{mbreceive}$ function is illustrated by figure 4.8 on page 56, which is a combination of selection (figure 4.8a) and removal (figure 4.8b).

4.3.2 Processes

In Erlang, all expressions are evaluated in lightweight *processes*. Each process has a globally unique identifier called a PID and a *mailbox* which buffers messages sent to it by other processes. A process' PID remains constant throughout its existence, while its mailbox changes over time as messages are sent and received.

Definition 4.3.1 (Process Identifier (PID))**.** *A Process Identifier (PID) ι is a globally unique identifier used to refer to a process. Not every PID is associated with a process, but each process must have a PID.*

4.4 Big-Step Operational Semantics

The big-step operational semantics can be used to reason about terminating computations of a single process (figure 4.12 on the following page). They relate a triple of expression e, environment ρ, and mailbox mb with a value v, potentially different environment ρ', and potentially modified mailbox mb'.

The rule Lit evaluates a literal *expression* to a literal *value* and Var looks up the value of v in environment ρ, returning it as a value. Nil and Cons are responsible for evaluating empty lists and cons cells respectively, and this is where we see the left-to-right evaluation order we have chosen for CoErl: the head of the cons cell is evaluated first, then the tail. Similar rules exist for tuples: TupleNil evaluates empty tuples and TupleCons evaluates tuples from left-to-right.

Note that when we evaluate these sub-expressions from left-to-right we use the original environment for each sub-expression, but we propagate the changed mailbox state with each computation.

To evaluate case expressions we use rule Case, which first evaluates the *argument* of the expression, then selects a clause, and finally evaluates the expression of the selected clause using the environment returned from the clause selection procedure. We use the environment returned by $\mathit{cselect}$ because it may contain variable bindings from the clause's pattern.

Function application is handled by Call, which evaluates the function $m : f/a$ in the environment ρ. First, we evaluate the arguments in left-to-right order, making sure to propagate

$$\frac{}{(l, \rho, mb) \Downarrow (l, \rho, mb)}\ \textsc{Lit} \qquad \frac{\rho(v) = l}{(v, \rho, mb) \Downarrow (l, \rho, mb)}\ \textsc{Var} \qquad \frac{}{([\,], \rho, mb) \Downarrow ([\,], \rho, mb)}\ \textsc{Nil}$$

$$\frac{(e_h, \rho, mb) \Downarrow (v_h, \rho_h, mb_h) \qquad (e_t, \rho, mb_h) \Downarrow (v_t, \rho_t, mb_t)}{([\, e_h \mid e_t \,], \rho, mb) \Downarrow ([\, v_h \mid v_t \,], \rho, mb_t)}\ \textsc{Cons}$$

$$\frac{}{(\{\,\}, \rho, mb) \Downarrow (\{\,\}, \rho, mb)}\ \textsc{TupleNil}$$

$$\frac{\begin{array}{c}(e_1, \rho, mb) \Downarrow (v_1, \rho_1, mb_1) \\ (e_2, \rho, mb_1) \Downarrow (v_2, \rho_2, mb_2) \qquad \ldots \qquad (e_n, \rho, mb_{n-1}) \Downarrow (v_n, \rho_n, mb_n)\end{array}}{(\{e_1, e_2, \ldots, e_n\}, \rho, mb) \Downarrow (\{v_1, v_2, \ldots, v_n\}, \rho, mb_n)}\ \textsc{TupleCons}$$

$$\frac{(e, \rho, mb) \Downarrow (v, \rho', mb') \qquad (e_b, \rho[x \mapsto v], mb') \Downarrow (v', \rho'', mb'')}{(\mathtt{let}\ x = e\ \mathtt{in}\ e_b, \rho, mb) \Downarrow (v', \rho, mb'')}\ \textsc{Let}$$

$$\frac{(e_1, \rho, mb) \Downarrow (v_1, \rho', mb') \qquad (e_2, \rho, mb') \Downarrow (v_2, \rho'', mb'')}{(e_1\ \mathtt{;;}\ e_2, \rho, mb) \Downarrow (v_2, \rho, mb'')}\ \textsc{Seq}$$

$$\frac{(e, \rho, mb) \Downarrow (v, \rho_e, mb') \qquad cselect(cs, v, \rho) = (e', \rho') \qquad (e', \rho', mb') \Downarrow (v', \rho'', mb'')}{(\mathtt{case}\ \langle e \rangle\ \mathtt{of}\ cs\ \mathtt{end}, \rho, mb) \Downarrow (v', \rho, mb'')}\ \textsc{Case}$$

$$\frac{\begin{array}{c}(e_{a_1}, \rho, mb) \Downarrow (v_{a_1}, \rho_1, mb_1) \\ \ldots \qquad (e_{a_n}, \rho, mb_(n-1) \Downarrow (v_{a_n}, \rho_n, mb_n) \qquad \rho(m, f, n) = \mathtt{fun}(a_1, a_2, \ldots, a_n)\ \rightarrow\ e\ \mathtt{end} \\ (e, \rho[a_1 \mapsto v_1, a_2 \mapsto v_2, \ldots, a_n \mapsto v_n], mb_n) \Downarrow (v, \rho', mb')\end{array}}{(\mathtt{call}\ m : f\ (e_{a_1}, \ldots, e_{a_n}), \rho, mb) \Downarrow (v, \rho, mb')}\ \textsc{Call}$$

$$\frac{receive(cs, mb, \rho) = \mathbf{ok}\ (e, \rho', mb') \qquad (e, \rho', mb') \Downarrow (v, \rho'', mb'')}{(\mathtt{receive}\ cs\ \mathtt{end}, \rho, mb) \Downarrow (v, \rho, mb'')}\ \textsc{Receive}$$

Figure 4.12: Big-Step Operational Semantics for CoErl

the mailbox state between each one. Then we apply the function itself, mapping each of the function's arguments to the corresponding value in the environment ρ.

The rule Receive handles `receive` expressions, which block unless a matching message can be found in the mailbox.

These operational semantics do not represent the full behaviour of a CoErl system and are instead meant to give a high-level overview of the behaviour of the language. Notably these semantics cannot model non-terminating computations or concurrent behaviour. Nonetheless they serve as a useful reference for how expressions are evaluated and how the state of the mailbox is carried between sub-expressions.

4.5 Small-Step Operational Semantics

This section presents a small-step operational semantics for the Core Erlang fragment. While a big-step semantics shows the order of evaluation and the propagation of mailbox state between sub-expressions, it does not offer insight into intermediate computational states, nor does it permit modelling of infinite computations.

Intermediate states and non-terminating computations are important concepts in Erlang: intermediate states allow for interruption of control flow so mailboxes can be appended to by other processes, and infinite computation is often used to model servers or other long-lived processes. In a pure language, infinite computations are not as useful as they are here: Erlang permits side effects through communication, so infinite computations may produce some useful result by the way they affect the evaluation of other processes.

The small-step semantics is modelled using an inductively defined binary relation represented by the $\rightarrow$ operator where both the left and right-hand sides of the relation are processes.

Due to Core Erlang's lexical scoping, it is necessary to add a scoping mechanism to the semantics which was not present in the big-step model. This is represented by σ in definitions of processes:

Definition 4.5.1 (Processes for small-step semantics). *In the small-step operational semantics, a process is a tuple of the form:*

$$(e, \rho, \sigma, mb, \iota)$$

Where e is either a value (prefixed by val) or an expression (without any prefix), ρ is the current context of variable bindings, σ is a stack of continuations, and mb and ρ are the mailbox and PID of the process respectively.

A continuation consists of an expression containing a hole ($\Box$) and a context ρ; together these can be used to resume some "outer" expression in a different context. For example, in the expression e_1 ;; e_2, both e_1 and e_2 must be evaluated using variable bindings from the current scope: no variables bound in e_1 should be visible in e_2, hence the current context must be captured and stored for when e_2 is later evaluated. Furthermore, once e_1 has been evaluated to some value, the process must have some way of knowing that e_2 must be evaluated next, so it is also necessary to store information about how to *continue* evaluation once a value has been computed.

The small-step rules in figure 4.13 on the next page and on page 62 represent the portion of CoErl which don't require a concurrent environment, i.e. where there are no other processes.

$$\frac{}{(l, \rho, \sigma, mb, \iota) \to (val\ l, \rho, \sigma, mb, \iota)}\ \textsc{Lit} \qquad \frac{}{(var\ x, \rho, \sigma, mb, \iota) \to (val\ \rho(x), \rho, \sigma, mb, \iota)}\ \textsc{Var}$$

$$\frac{}{([\,], \rho, \sigma, mb, \iota) \to (val\ [\,], \rho, \sigma, mb, \iota)}\ \textsc{Nil}$$

$$\frac{}{([\, e_h \mid e_t \,], \rho, \sigma, mb, \iota) \to (e_h, \rho, ([\, \Box \mid e_t \,], \rho) :: \sigma, mb, \iota)}\ \textsc{Cons1}$$

$$\frac{}{(val\ v, \rho, ([\, \Box \mid e_t \,], \rho') :: \sigma, mb, \iota) \to (e_t, \rho', ([\, val\ v \mid \Box \,], \rho') :: \sigma, mb, \iota)}\ \textsc{Cons2}$$

$$\frac{}{(val\ v_t, \rho, ([\, val\ v_h \mid \Box \,], \rho') :: \sigma, \iota) \to (val\ [\, v_h \mid v_t \,], \rho', \sigma, \iota)}\ \textsc{ConsVal}$$

$$\frac{}{(\{\,\}, \rho, \sigma, mb, \iota) \to (val\ \{\,\}, \rho, \sigma, mb, \iota)}\ \textsc{TupleNil}$$

$$\frac{}{(\{e_1, e_2, \ldots, e_n\}, \rho, \sigma, mb, \iota) \to (e_1, \{\Box, e_2, \ldots, e_n\}, \rho) :: \sigma, mb, \iota)}\ \textsc{TupleN}$$

$$\frac{}{\begin{array}{c}(val\ v_m, \rho, (\{val\ v_1, \ldots, val\ v_{m-1}, \Box, e_{m+1}, \ldots\}, \rho') :: \sigma, mb, \iota) \to \\ (e_{m+1}, \rho', (\{val\ v_1, \ldots, val\ v_{m-1}, val\ v_m, \Box, \ldots\}, \rho') :: \sigma, mb, \iota)\end{array}}\ \textsc{TupleStep}$$

$$\frac{}{(val\ v_n, \rho, (\{val\ v_1, val\ v_2, \ldots, \Box\}, \rho') :: \sigma, mb, \iota) \to (val\ \{v_1, v_2, \ldots, v_n\}, \rho', \sigma, mb, \iota)}\ \textsc{TupleVal}$$

$$\frac{}{(\mathsf{let}\ x = e1\ \mathsf{in}\ e2, \rho, \sigma, mb, \iota) \to (e1, \rho, (\mathsf{let}\ x = \Box\ \mathsf{in}\ e2, \rho) :: \sigma, mb, \iota)}\ \textsc{Let1}$$

$$\frac{}{(val\ v, \rho, (\mathsf{let}\ x = \Box\ \mathsf{in}\ e2, \rho') :: \sigma, mb, \iota) \to (e2, \rho'[x \mapsto v], \sigma, mb, \iota)}\ \textsc{Let2}$$

$$\frac{}{(e1\ ;;\ e2, \rho, \sigma, mb, \iota) \to (e1, \rho, (\Box\ ;;\ e2, \rho) :: \sigma, mb, \iota)}\ \textsc{Seq1}$$

$$\frac{}{(val\ v, \rho, (\Box\ ;;\ e2, \rho') :: \sigma, mb, \iota) \to (e2, \rho', \sigma, mb, \iota)}\ \textsc{Seq2}$$

$$\frac{}{(\mathsf{case}\ \langle e \rangle\ \mathsf{of}\ cs\ \mathsf{end}, \rho, \sigma, mb, \iota) \to (e, \rho, (\mathsf{case}\ \langle \Box \rangle\ \mathsf{of}\ cs\ \mathsf{end} :: \sigma, mb, \iota)}\ \textsc{Case1}$$

$$\frac{cselect(cs, v, \rho') = \mathbf{ok}\ (e, \rho_e)}{(val\ v, \rho, (\mathsf{case}\ \langle \Box \rangle\ \mathsf{of}\ cs\ \mathsf{end}, \rho') :: \sigma, mb, \iota) \to (e, \rho_e, \sigma, mb, \iota)}\ \textsc{Case2}$$

$$\frac{}{\begin{array}{c}(\mathsf{call}\ m : f\ (e_{a_1}, \ldots, e_{a_n}), \rho, \sigma, mb, \iota) \to \\ (e_{a_1}, \rho, (\mathsf{call}\ m : f\ (\Box, \ldots, e_{a_n}), \rho) :: \sigma, mb, \iota)\end{array}}\ \textsc{Call1}$$

$$\frac{}{\begin{array}{c}(val\ e_{a_m}, \rho, (\mathsf{call}\ m : f\ (val\ v_{a_1}, \ldots, val\ v_{a_{m-1}}, \Box, e_{a_{m+1}}, \ldots), \rho') :: \sigma, mb, \iota) \to \\ (e_{a_{m+1}}, \rho', (\mathsf{call}\ m : f\ (val\ v_{a_1}, \ldots, val\ v_{a_{m-1}}, val\ v_{a_m}, \Box, \ldots), \rho') :: \sigma, mb, \iota)\end{array}}\ \textsc{CallStep}$$

$$\frac{\rho(m, f, n) = \mathsf{fun}(a_1, \ldots, a_n)\ \to\ e_f\ \mathsf{end}}{\begin{array}{c}(val\ e_{a_n}, \rho, (\mathsf{call}\ m : f\ (val\ v_{a_1}, \ldots, \Box), \rho') :: \sigma, mb, \iota) \to \\ (e_f, \rho[a_1 \mapsto v_{a_1}, \ldots, a_n \mapsto v_{a_n}], \sigma, mb, \iota)\end{array}}\ \textsc{CallVal}$$

Figure 4.13: Small-Step semantics for single CoErl processes (part 1 of 2)

$$\frac{mbreceive(cs, mb, \rho) = \textbf{ok}\ (e, \rho_e, mb')}{(\texttt{receive cs end}, \rho, \sigma, mb, \iota) \to (e, \rho_e, \sigma, mb', \iota)}\ \textsc{Receive}$$

$$\frac{}{(e_1\ !\ e_2, \rho, \sigma, mb, \iota) \to (e_1, \rho, (\Box\ !\ e_2, \rho) :: \sigma, mb, \iota)}\ \textsc{Send1}$$

$$\frac{}{(val\ \iota, \rho, (\Box\ !\ e_2, \rho') :: \sigma, mb, \iota) \to (e_2, \rho', (\iota\ !\ \Box, \rho') :: \sigma, mb, \iota)}\ \textsc{Send2}$$

$$\frac{}{(\texttt{self}, \rho, \sigma, mb, \iota) \to (val\ \iota, \rho, \sigma, mb, \iota)}\ \textsc{Self}$$

Figure 4.13: Small-Step semantics for single CoErl processes (part 2 of 2)

The Lit and Var rules are responsible for transforming expressions that are literals and variable names respectively into values.

Empty lists and empty tuples are handled by Nil and TupleNil respectively. When we encounter non-empty compound types however, we use continuations. For example, when a cons cell is first encountered (rule Cons) a continuation $[\ \Box \mid e_t\ ]$ and environment ρ is pushed onto σ so we can return to evaluating the tail later on. Once the head has been evaluated we then evaluate the tail (rule Cons2), saving the value we computed from the head expression. Finally, once both the head and tail have been fully evaluated, we create a cons *value* and populate it with the head value v_h and tail value v_t (rule ConsVal).

Similar rules exist for tuples (rules TupleCons, TupleStep, and TupleVal). The main difference is the TupleStep rule which iterates to the next element in a tuple unless it is the final element: we save the value we have computed in the continuation and start evaluating the next sub-expression in the tuple. At each step we restore the original environment ρ' from the continuation, discarding any changes made in the previous sub-expression.

Let1 and Let2 are responsible for evaluating let bindings using a continuation style: Let1 saves the clauses of the `let` expression so they can be returned to later (by pushing a continuation onto σ where $\Box$ is the hole for the value computed by the current expression) and Let2 resumes evaluation of a `let` expression once the argument has been completely evaluated.

Function calls are handled similarly to tuples: when they are first encountered we create a continuation and start evaluating the arguments from left-to-right (Call1 and CallStep). Once all arguments have been evaluated we look up the function in environment ρ, bind all arguments to the values we have computed, and start evaluating the function definition (CallVal).

The rule Receive handles `receive` expressions where there is a message in the mailbox which can be received, otherwise the process becomes stuck.

$$\frac{}{\begin{gathered}(\mathit{val}\, v, \rho_\iota, (\iota_2 \,!\, \Box, \rho'_\iota) :: \sigma_\iota, mb_\iota, \iota) || (e_{\iota_2}, \rho_{\iota_2}, \sigma_{\iota_2}, mb_{\iota_2}, \iota_2) \rightarrow \\ (\mathit{val}\, v, \rho'_\iota, \sigma_\iota, mb_\iota, \iota) || (e_{\iota_2}, \rho_{\iota_2}, \sigma_{\iota_2}, mb_{\iota_2} \mathbin{++} [v], \iota_2)\end{gathered}}\ \Pi\textsc{Send1}$$

$$\frac{}{(\mathit{val}\, v, \rho, (\iota \,!\, \Box, \rho') :: \sigma, mb, \iota) \rightarrow (\mathit{val}\, v, \rho', \sigma, mb \mathbin{++} [v], \iota)}\ \Pi\textsc{Send2}$$

$$\frac{\nexists p \in \Pi.\, \mathit{pid}(p) = \iota'}{(\mathit{val}\, v, \rho, (\iota' \,!\, \Box, \rho') :: \sigma, mb, \iota) || \Pi \rightarrow (\mathit{val}\, v, \rho', \sigma, mb, \iota) || \Pi}\ \Pi\textsc{Send3}$$

Figure 4.14: Small-Step send semantics for CoErl

The rules Send1 and Send2 are responsible for evaluating send expressions but *not the concurrent behaviour* of them. Rule Send1 evaluates the left-hand side of the send operator and Send2 evaluates the right-hand side, but note that the process then becomes stuck. Finally, Self gives a process access to its own PID.

4.5.1 Sending Messages

To actually send messages in CoErl a concurrent environment is needed, i.e. one or more processes running concurrently. The parallel composition operator $||$ is used to represent two or more processes running concurrently: $p||q$ represents process p running concurrently with process q. The order of processes does not matter: $p||q$ is the same as $q||p$, and $p||(q||r)$ is the same as $q||(r||p)$, etc.

The rules in figure 4.14 operate on these environments and are responsible for sending messages between processes. There are three possible scenarios when sending messages in a CoErl system:

- Process ι sends a message v to process ι', and ι' is part of the system. The message is appended to the mailbox of ι' and process ι continues executing (ΠSend1).
- Process ι sends a message to *itself*, appending the message to its own mailbox (ΠSend2).
- Process ι sends a message to ι' which *is not* in the concurrent environment, so the message "disappears" and ι continues to execute (ΠSend3).

Note in ΠSend1 that a message can be sent to ι' regardless of its state: the evaluation of ι' can be interrupted at any time when delivering a message, and ι' cannot affect whether or not the message is delivered. These rules replicate Erlang's behaviour: messages can be sent to any process asynchonously, sends to non-existent processes fail silently, and processes can send messages to themselves.

$$\frac{p \rightarrow p'}{p \| q \rightarrow p' \| q}\ \Pi\textsc{Step1} \qquad\qquad \frac{p \rightarrow p' \qquad q \rightarrow q'}{p \| q \rightarrow p' \| q'}\ \Pi\textsc{Step2}$$

$$\begin{gathered} p \| q \equiv q \| p \\ p \| (q \| r) \equiv (p \| q) \| r \end{gathered}$$

Figure 4.15: Small-Step concurrent semantics for CoErl

$$\frac{}{p \rightarrow^* p}\ \textsc{Refl} \qquad\qquad \frac{p \rightarrow p' \qquad p' \rightarrow^* p''}{p \rightarrow^* p''}\ \textsc{Trans}$$

Figure 4.16: Reflexive transitive closure for CoErl small-step semantics

Although these rules are effectively a synchronisation point between two processes, it is important to note that the *expressions* are not synchronised: the expression of one process is affecting the state of another process and does *not* rely on its current expression.

4.5.2 Concurrent Processes

We must also allow evaluation of processes in a concurrent system even when they are not sending messages to each other. The rules in figure 4.15 permit this behaviour: ΠStep1 allows a single process to make a step. Repeated applications of this rule represent concurrency through interleaved execution. On the other hand, ΠStep2 allows us to imitate Erlang's behaviour on multiple CPU cores, where two processes *actually* execute at the same time.

4.5.3 Reflexive Transitive Closure

The final part of the small-step semantics is the reflexive transitive closure, shown in figure 4.16. We use this operator ($\rightarrow^*$) to repeatedly apply the small-step relation $\rightarrow$ zero or more times in succession. This allows us to represent computations which require more than a single step, and finitely approximate the behaviour of non-terminating computations.

At this point there is a correspondence between the big-step semantics in figure 4.12 and the small-step semantics presented in this section: all computations which can be represented in the big-step semantics can be represented in the small-step semantics:

$$\forall (e, mb, \rho), (e', mb', \rho').\, (e, mb, \rho) \Downarrow (e', mb', \rho') \implies \exists \sigma', \iota.\, (e, mb, [\,], \iota) \rightarrow^* (e', mb', \sigma', \iota)$$

Unfortunately the big-step semantics does not allow us to interrupt the evaluation of an expression so messages cannot be appended to mailboxes during execution, and the big-step semantics cannot model non-terminating computations.

4.6 Infinite Computations

Real world Erlang/OTP systems often contain processes which run ad infinitum. These non-terminating processes can be useful because they can have side effects in the form of communication with other processes. This is not to say that these processes cannot terminate, rather that there is no guarantee that they will. For example, the counter server seen earlier in listing 6a on page 22 will terminate if it receives the `'stop'` message, but it will otherwise loop forever.

The big-step semantics cannot represent non-terminating computations directly because it associates a process configuration with a *final* state, i.e. a configuration containing a value instead of an expression. One option to address this shortcoming is to introduce a counter which decrements as evaluation progresses, until either the program terminates or the counter reaches zero. This can be done in two ways: by modifying programs, or by modifying the big-step relation. If we introduce a counter directly into programs, the non-terminating program will necessarily terminate after a finite number of steps, but care must be taken to ensure equivalence with the original program, modulo termination. On the other hand, adding a decrementing counter to the relation itself allows for finite approximation of non-terminating programs without making changes to the program itself.

On the other hand, the small-step semantics can approximate the behaviour of infinite computations without modification. The reflexive transitive closure (figure 4.16 on the previous page) does not only associate a process configuration with its final state, but also *every* intermediate state. Therefore, we can approximate the behaviour of a non-terminating process by applying rule Trans a finite number of times, stopping computation by rule Refl.

Chapter 5

Behavioural Analysis of CoErl via Traces

As the small-step relation $\rightarrow$ from the previous chapter implements Erlang's communication model, it can be used to reason about the behaviour of mailboxes over time. Several rules in the small-step semantics affect the mailbox of one or more processes: the send rules append messages to mailbox of processes while the receive rules remove them. While the semantics can be used to reason about communication in an Erlang system by repeated application of the small-step relation via the reflexive transitive closure, it is a somewhat tedious process. At each step the *entire* state of *all* processes must be considered, including the current variable assignments and continuation stack.

To make it easier to reason about communication, this chapter presents a useful abstraction over the small-step semantics in the form of an *Labelled Transition System* (LTS). LTSs abstract over the behaviour of a system by *tracing* the events which occur when transitions (i.e. steps) in that system are made. By carefully choosing the events which are traced in an LTS, it is possible to create a separation of concern where events deemed to be "interesting" are traced and the rest of the system's behaviour is hidden beneath the abstraction of the traces.

Labelled Transition System (LTS) are often used to reason about the *observable* behaviour of a program, for example to determine whether a refactored version of a program behaves identically, or performs the same I/O operations as it did before the refactoring. For the purposes of reasoning about communication in CoErl, the definition of an LTS seems ideal: by carefully labelling rules of the small-step semantics, mailbox behaviour can be isolated from internal process behaviour such as the variable binding steps and the pushing/popping of continuations.

This chapter contributes a trace-based analysis of CoErl which allows us to reason about communications separately from the internal behaviour of the concurrent systems being modelled. We can use this approach to identify communication discrepancies in CoErl without using the operational semantics directly. Furthermore, in order to reason about Erlang's unique asynchronous and out-of-order communication model we introduce an *arrival* label which allows us to view message arrivals separately from receives, in contrast to other channel-based languages and process calculi.

Overview This chapter makes heavy use of *Labelled Transition Systems* (LTSs), which are described in section 2.2 on page 26. We start wich a labelled version of the small-step operational semantics from section 4.5, where we add labels to each transition of the relation which describe the communicating behaviour of individual processes (section 5.1.2) and concurrent CoErl systems (section 5.2). These sections also introduce the concepts of *traces*: sequences of events which describe the communicating behaviour of processes and systems

We then define equivalence between traces in both strong and weak forms, where the latter allows us to compare the communicating behaviour of two processes or systems modulo non-communicating transitions (section 5.3). Afterwards, we look at how the asynchronous behaviour of Erlang communications can be observed in traces, namely that an arrival of a message can sometimes take place earlier during execution without affecting evaluation (section 5.4). This is followed by a method of replaying traces which allows us to simulate the behaviour of a process' mailbox using its trace (section 5.5), and a techinque for detecting orphan messages in a trace (section 5.6). Finally, we discuss how these analyses can be applied to infinite computations (section 5.7).

5.1 Labelled Small-Step Semantics

We can already represent the small-step semantics (section 4.5 on page 60) as a transition system using definition 2.2.1 on page 26:

$$(S, \rightarrow)$$

where S is the set of all possible *process configurations* and $\rightarrow$ is the binary small-step relation over some subset of all possible process configurations.

Definition 5.1.1 (Process configurations)**.** *The set of process configurations S is the set of all possible process states consisting of all possible expressions or values (ev), all possible environments ($[var \times val]$), all possible continuation states (σ), all possible mailboxes (mb), and all possible PIDs (ι):*

$$S = ev \times (var \mapsto val) \times \sigma \times mb \times \iota$$

In order to provide an LTS for CoErl, two elements are required:

1. a set of labels (A); and
2. a relation $R \subseteq (S \times A \times S)$

Both definitions require careful consideration: the chosen labels should be fit for the purposes of reasoning about communications in CoErl, and the relation should be a faithful representation of the small-step semantics.

Choosing labels suitable for the analysis depends on exactly what the aim of the analysis is: if labels capture too much information it could be tedious to reason about the desired properties, and if labels do not capture enough information then it might be impossible to reason about interesting behaviour. For the analysis of CoErl, the labels should capture all effects on the mailbox (arrivals and receiving of messages), but not too broad as to also capture irrelevant parts of the small-step semantics, again making the analysis tedious. One measure of expressiveness of the labels is to ensure that any "replay" of a label on a mailbox results in the same mailbox as seen during a transition in the LTSs. For example, given a function $\mathtt{apply}(\alpha, \mathtt{mb})$ which applies the "effect" of label α to a mailbox $\mathtt{mb}$, then the following should hold:

$$\forall (p, \alpha, p') \in R.\, \mathtt{apply}(\alpha, \mathtt{mailbox}(p)) = \mathtt{mailbox}(p')$$

As for the relation R, it can be considered faithful to the original small-step semantics if it is both sound and complete with respect to them. The new relation R will be considered *complete* if there is a labelled equivalent of every transition from the small-step semantics in R:

Definition 5.1.2 (Completeness)**.**

$$\forall (p, p') \in (S \times S) : p \to p'.\, (\exists \alpha \in A.\, (p, \alpha, p') \in R)$$

and it will be considered *sound* if every possible transition in R is also present in the original small-step relation, albeit without labels:

Definition 5.1.3 (Soundness)**.**

$$\forall (p, \alpha, p') \in R.\, p \to p'$$

5.1.1 Labels

In CoErl, there are two ways of affecting a mailbox in some part of the system, both of which are syntax directed, via the following expressions:

1. $\iota \,!\, \nu$: appends the message ν to the end of the mailbox of the process with PID ι (if it exists)

2. `receive` `cs` `end`: removes the first message that matches clauses `cs` from the process' own mailbox *if possible*[1]

As such the first two labels for the LTS will mirror the effects of these expressions:

1. $\iota \,!\, v$: the message v is sent to process ι

2. $?\, v$ the message v is received from process' own mailbox

When analysing processes in a *closed* system[2], these send and receive labels are sufficient to reason about the state of all mailboxes in the system. When only observing *part* of a system however, these two labels are not sufficient to capture all mailbox behaviour. For example, when observing a single process in a larger system, any other process in the system could send a message to the observed process, and no send or receive label would be observed. Ultimately, the problem is one of perspective: observing the send and receive behaviour of a single process does not give full insight into the state of that process' mailbox *as other processes can send other messages to it*. Therefore, a third label is required which represents this behaviour:

3. $\mathsf{arr}\; v$: the message v arrives in the process' own mailbox

Now, observing the events which occur when observing a single process will represent all behaviour of that process' mailbox.

Finally, consider that the final LTS will be of the form (S, A, R) where $R \subseteq (S \times A \times S)$, which means that *every* transition in the system must be labelled. Also consider that not *every* rule of the small-step semantics relates to communication, for example variable assignment and lookup. To meet both requirements, a final label is introduced which represents the "internal" behaviour of a process, i.e. transitions which do not directly affect the state of a mailbox. In the small-step semantics, rules such as variable assignment and lookup can be considered internal as they do not directly manipulate a mailbox. This label is written τ:

4. τ: the process makes an internal transition

Together, these 4 labels complete the definition of A for an LTS which captures the mailbox behaviour of CoErls small-step semantics:

[1]will block otherwise

[2]A closed system is one where there are no external sources of messages and it is always known whether or not a given PID is associated with a process.

Definition 5.1.4. [*Labels*]

$$A ::= \iota \,!\, v \mid ?\, v \mid \mathsf{arr}\, v \mid \tau$$

A label is either a send of message v *to process* ι $(\iota \,!\, v)$*, a receive of message* v *from the mailbox* $(?\, v)$*, arrival of message* v *at the end of the mailbox* $(\mathsf{arr}\, v)$ *or an internal action unrelated to communication* (τ)*.*

With this, the definitions of S (configurations) and A (labels) are complete. The final element is R, which will be a labelled version of the small-step semantics with equivalent behaviour to the original.

5.1.2 Labelled Small-Step Relation

The small-step relation from section 4.5 relates two process configurations via the $\rightarrow$ relation. With respect to process configurations, the small-step semantics relates two process configurations such that $(\rightarrow) \subseteq (S \times S)$. The definition of LTSs requires that a relation of the form $R \subseteq (S \times A \times R)$ is used, where A is a set of labels (definition 2.2.3). The most straightforward way to transform $\rightarrow$ into a labelled equivalent is to add the label τ to each rule of the relation. This approach does not capture the behaviour of mailboxes because the following previously given property will not hold:

$$\forall (p, \alpha, p') \in R.\, \mathsf{apply}(\alpha, \mathsf{mailbox}(p)) = \mathsf{mailbox}(p')$$

By simply labelling each rule of the small-step semantics with τ, mailbox states cannot be reconstructed by "replaying" events. In general, though, this tactic can be used for *most* rules in the small-step semantics as long as special consideration is given to communicating syntactic constructs.

A labelled version of the small-step relation is $\xrightarrow{\alpha}$, where $\alpha \in A$ is a label. Figure 5.1 on the next page shows this relation, which relates two configurations of a single process: it does not encapsulate the concurrent behaviour of the language, which will be added later.

All rules are labelled with τ except for ASend, AReceive, and AArrive. The ASend rule is labelled with $\iota' \,!\, v$, where ι' and v are derived from the process' state. In AReceive, the label is determined by the value of the message v removed from the mailbox. For the purposes of this rule, assume that $\mathsf{remove}(v, \mathsf{mb})$ is mb with the first instance of value v removed. Finally, the rule AArrive allows messages to be arbitrarily appended to the mailbox of the process. From the perspective of a single process, this rule allows for non-deterministic behaviour derived from the mailbox, wherein a process may be sent messages by other processes without being aware of

$$\frac{}{(l, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (\mathit{val}\ l, \rho, \sigma, mb, \iota)}\ \textsc{ALit} \qquad \frac{}{(\mathit{var}\ x, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (\mathit{val}\ \rho(x), \rho, \sigma, mb, \iota)}\ \textsc{AVar}$$

$$\frac{}{(\texttt{[]}, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (\mathit{val}\ \texttt{[]}, \rho, \sigma, mb, \iota)}\ \textsc{ANil}$$

$$\frac{}{(\texttt{[}\, e_h \mid e_t \,\texttt{]}, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (e_h, \rho, (\texttt{[}\, \Box \mid e_t \,\texttt{]}, \rho) :: \sigma, mb, \iota)}\ \textsc{ACons1}$$

$$\frac{}{(\mathit{val}\ v, \rho, (\texttt{[}\, \Box \mid e_t \,\texttt{]}, \rho') :: \sigma, mb, \iota) \xrightarrow{\tau} (e_t, \rho', (\texttt{[}\, \mathit{val}\ v \mid \Box \,\texttt{]}, \rho') :: \sigma, mb, \iota)}\ \textsc{ACons2}$$

$$\frac{}{(\mathit{val}\ v_t, \rho, (\texttt{[}\, \mathit{val}\ v_t \mid \Box \,\texttt{]}, \rho') :: \sigma, \iota) \xrightarrow{\tau} (\mathit{val}\ \texttt{[}\, v_h \mid v_t \,\texttt{]}, \rho', \sigma, \iota)}\ \textsc{AConsVal}$$

$$\frac{}{(\{\,\}, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (\mathit{val}\ \{\,\}, \rho, \sigma, mb, \iota)}\ \textsc{ATupleNil}$$

$$\frac{}{(\{e_1, e_2, \ldots, e_n\}, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (e_1, \{\Box, e_2, \ldots, e_n\}, \rho) :: \sigma, mb, \iota)}\ \textsc{ATupleN}$$

$$\frac{}{\begin{array}{c}(\mathit{val}\ v_m, \rho, (\{\mathit{val}\ v_1, \ldots, \mathit{val}\ v_{m-1}, \Box, e_{m+1}, \ldots\}, \rho') :: \sigma, mb, \iota) \xrightarrow{\tau} \\ (e_{m+1}, \rho', (\{\mathit{val}\ v_1, \ldots, \mathit{val}\ v_{m-1}, \mathit{val}\ v_m, \Box, \ldots\}, \rho') :: \sigma, mb, \iota)\end{array}}\ \textsc{ATupleStep}$$

$$\frac{}{(\mathit{val}\ v_n, \rho, (\{\mathit{val}\ v_1, \mathit{val}\ v_2, \ldots, \Box\}, \rho') :: \sigma, mb, \iota) \xrightarrow{\tau} (\mathit{val}\ \{v_1, v_2, \ldots, v_n\}, \rho', \sigma, mb, \iota)}\ \textsc{ATupleVal}$$

$$\frac{}{(\texttt{let}\ x = e1\ \texttt{in}\ e2, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (e1, \rho, (\texttt{let}\ x = \Box\ \texttt{in}\ e2, \rho) :: \sigma, mb, \iota)}\ \textsc{ALet1}$$

$$\frac{}{(\mathit{val}\ v, \rho, (\texttt{let}\ x = \Box\ \texttt{in}\ e2, \rho') :: \sigma, mb, \iota) \xrightarrow{\tau} (e2, \rho'[x \mapsto v], \sigma, mb, \iota)}\ \textsc{ALet2}$$

$$\frac{}{(e1\ \texttt{;;}\ e2, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (e1, \rho, (\Box\ \texttt{;;}\ e2, \rho) :: \sigma, mb, \iota)}\ \textsc{ASeq1}$$

$$\frac{}{(\mathit{val}\ v, \rho, (\Box\ \texttt{;;}\ e2, \rho') :: \sigma, mb, \iota) \xrightarrow{\tau} (e2, \rho', \sigma, mb, \iota)}\ \textsc{ASeq2}$$

$$\frac{}{(\texttt{case}\ \langle e \rangle\ \texttt{of}\ cs\ \texttt{end}, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (e, \rho, (\texttt{case}\ \langle \Box \rangle\ \texttt{of}\ cs\ \texttt{end} :: \sigma, mb, \iota)}\ \textsc{ACase1}$$

$$\frac{\mathit{cselect}(cs, v, \rho') = \mathbf{ok}\ (e, \rho_e)}{(\mathit{val}\ v, \rho, (\texttt{case}\ \langle \Box \rangle\ \texttt{of}\ cs\ \texttt{end}, \rho') :: \sigma, mb, \iota) \xrightarrow{\tau} (e, \rho_e, \sigma, mb, \iota)}\ \textsc{ACase2}$$

Figure 5.1: Labelled small-step operational semantics for a single process (part 1 of 2)

$$\frac{\mathit{mbreceive}(cs, mb, \rho) = \textbf{ok}\ (e, \rho_e, mb') \qquad mb' = \mathit{remove1}(v, mb)}{(\texttt{receive cs end}, \rho, \sigma, mb, \iota) \xrightarrow{?\,v} (e, \rho_e, \sigma, mb', \iota)}\ \textsc{AReceive}$$

$$\frac{}{(e_1\ !\ e_2, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (e_1, \rho, (\Box\ !\ e_2, \rho) :: \sigma, mb, \iota)}\ \textsc{ASend1}$$

$$\frac{}{(\mathit{val}\ \iota, \rho, (\Box\ !\ e_2, \rho') :: \sigma, mb, \iota) \xrightarrow{\tau} (e_2, \rho', (\iota\ !\ \Box, \rho') :: \sigma, mb, \iota)}\ \textsc{ASend2}$$

$$\frac{}{(\texttt{self}, \rho, \sigma, mb, \iota) \xrightarrow{\tau} (\mathit{val}\ \iota, \rho, \sigma, mb, \iota)}\ \textsc{ASelf}$$

$$\frac{}{(\mathit{val}\ v, \rho, (\iota'\ !\ \Box, \rho') :: \sigma, mb, \iota) \xrightarrow{\iota'\,!\,v} (\mathit{val}\ v, \rho', \sigma, mb, \iota)}\ \textsc{ASend}$$

$$\frac{}{(e, \rho, \sigma, mb, \iota) \xrightarrow{\mathit{arr}\ v} (e, \rho, \sigma, mb \mathbin{++} v, \iota')}\ \textsc{AArrive}$$

Figure 5.1: Labelled small-step operational semantics for a single process (part 2 of 2)

the existence of these processes. This non-deterministic behaviour becomes more constrained in the concurrent small-step semantics, where the transition is only permitted when there is a corresponding send label from another process.

5.1.3 Labelled Transition System

The labelled version of the small-step semantics can be used to complete the definition of a labelled transition system which represents the behaviour of individual CoErl processes. As the labelled small-step semantics relates two process configurations via a label, it is a subset of $(S \times A \times R)$, which means it can be used as the relation for an LTS. By using the definitions of S and A as per the earlier unlabelled transition system, the labelled small-step semantics form the following LTS:

$$(S, A, \xrightarrow{\alpha})$$

5.1.4 Reflexive Transitive Closure

The reflexive transitive closure of the labelled small-step semantics is similar to the unlabelled version (section 4.5.3). The small-step $\xrightarrow{\alpha}$ relates two process configurations via a single label, representing a single step of computation. As the reflexive transitive closure relates a two process configurations via zero or more single steps of computation, it will be labelled with a sequence of zero or more labels, representing the order in which the labels occurred.

This sequence of labels forms a trace of events, representing the labelled transitions taken by a process to reach its new configuration:

Definition 5.1.5 (Trace of individual processes)**.** *A trace of an individual process has the syntax of* T*, defined as follows:*

$$\mathsf{T} ::= \epsilon \mid \alpha.\mathsf{T}$$

where ϵ *is the neutral element representing an empty trace, while* $\alpha.\mathsf{T}$ *represents the label* $\alpha \in \mathsf{A}$ *preceding the trace* T*.*

The $\xrightarrow{\mathsf{T}}{}^*$ relation in figure 5.2 on the following page is the labelled reflexive transitive closure for individual processes, consisting of two rules: either the process takes zero steps and produces an empty trace (rule Refl), or it takes zero or more steps producing a non-empty trace (rule Trans). In the transitive case, the label from a single step (via $\xrightarrow{\alpha}$) is concatenated with the trace from the following steps.

Each trace T describes the sequence of transitions taken by a process during computation, such that n repeated applications of the labelled small-step semantics of the form:

$$p_1 \xrightarrow{\alpha_1} p_2, p_2 \xrightarrow{\alpha_2} p_3, \dots, p_n \xrightarrow{\alpha_n} p_{n+1}$$

yield the following trace:

$$\alpha_1.\alpha_2.\dots.\alpha_n.\epsilon$$

The reflexive transitive closure serves as a convenient method for generating traces of processes, where the Trans rule generates a head and tail of a trace, and Refl terminates a trace with the neutral element. As such, the trace above can be derived via repeated application of these rules:

$$\dfrac{p_1 \xrightarrow{\alpha_1} p_2 \qquad \dfrac{p_2 \xrightarrow{\alpha_2} p_3 \qquad \begin{array}{c}\dfrac{p_n \xrightarrow{\alpha_n} p_{n+1} \qquad \dfrac{}{p_{n+1} \xrightarrow{\epsilon}{}^* p_{n+1}}\,\textsc{Refl}}{p_n \xrightarrow{\alpha_n.\epsilon}{}^* p_{n+1}}\,\textsc{Trans} \\ \vdots\end{array}}{p_2 \xrightarrow{\alpha_2.\dots.\alpha_n.\epsilon}{}^* p_{n+1}}\,\textsc{Trans}}{p_1 \xrightarrow{\alpha_1.\alpha_2.\dots.\alpha_n.\epsilon}{}^* p_{n+1}}\,\textsc{Trans}$$

5.2 Concurrent Labelled Small-Step Semantics

The labelled small-step semantics from the previous section only describes the behaviour of individual processes, with the notable caveat that *any* message can arrive in a process' mailbox at

$$\frac{}{p \xrightarrow{\epsilon}{}^{*} p}\ \textsc{Refl} \qquad\qquad \frac{p \xrightarrow{\alpha} p' \qquad p' \xrightarrow{T}{}^{*} p''}{p \xrightarrow{\alpha.T}{}^{*} p''}\ \textsc{Trans}$$

Figure 5.2: Reflexive transitive closure of the labelled small-step semantics

any time. With respect to an individual CoErl process, the mailbox is a source of non-determinism as the evaluation of receive expressions depends on the mailbox state, and the mailbox state can be modified by other processes.

In this section, a concurrent semantics is given for CoErl, using the labelled small-step semantics from the previous section as its foundation. Figure 5.3 on the next page relates two concurrent system configurations via the $\overset{\iota}{\underset{\alpha}{\twoheadrightarrow}}$ relation, where each transition is labelled with a PID ι and a label α. A concurrent system configuration represents the state of one or more processes which are running together:

Definition 5.2.1 (Concurrent system configuration). *A concurrent system configuration* (S^{Π}) *is one or more process configurations* ($p, q, r, \ldots \in S$, *definition 5.1.1*)*:*

$$S^{\Pi} = p \parallel q \parallel \ldots$$

Furthermore, the symbol Π *is used to represent zero or more processes.*

These configurations should be considered equal regardless of the order in which process configurations appear within them, such that "p and q running concurrently" is no different from "q and p running concurrently". Therefore, configurations can be considered as *sets* of process configurations where no PID occurs more than once. With regards to notation, this means that two system configurations should be considered equivalent if they contain the same process configurations, regardless of order:

Definition 5.2.2 (Associativity and commutativity of configurations). *For all* $p, q, r \in S$ *:*

$$p \parallel q \equiv q \parallel r$$
$$p \parallel (q \parallel r) \equiv (p \parallel q) \parallel r$$

The rules in figure 5.3 on the following page relate two system configurations by a label α and a PID ι, where $x \overset{\iota}{\underset{\alpha}{\twoheadrightarrow}} y$ represents a transition from configuration x to y by the process with PID making transition α.

The first rule of the concurrent semantics is ΠInternal which permits any process in the system to freely make an internal transition, allowing for processes to evaluate in any order

$$\frac{p \xrightarrow{\tau} p' \qquad \mathtt{pid}(p) = \iota}{p \parallel \Pi \xrightarrow[\tau]{\iota}\!\!\!\!\rightarrow p' \parallel \Pi}\ \Pi\textsc{Internal} \qquad\qquad \frac{p \xrightarrow{?\,v} p' \qquad \mathtt{pid}(p) = \iota}{p \parallel \Pi \xrightarrow[?\,v]{\iota}\!\!\!\!\rightarrow p' \parallel \Pi}\ \Pi\textsc{Receive}$$

$$\frac{p \xrightarrow{\iota'\,!\,v} p' \qquad q \xrightarrow{\mathtt{arr}\ v} q' \qquad \mathtt{pid}(p) = \iota \qquad \mathtt{pid}(q) = \iota'}{p \parallel q \parallel \Pi \xrightarrow[\iota'\,!\,v]{\iota}\!\!\!\!\rightarrow p' \parallel q' \parallel \Pi}\ \Pi\textsc{Send1}$$

$$\frac{p \xrightarrow{\iota\,!\,v} p' \qquad p' \xrightarrow{\mathtt{arr}\ v} p'' \qquad \mathtt{pid}(p) = \iota}{p \parallel \Pi \xrightarrow[\iota'\,!\,v]{\iota}\!\!\!\!\rightarrow p'' \parallel \Pi}\ \Pi\textsc{Send2}$$

$$\frac{p \xrightarrow{\iota'\,!\,v} p' \qquad \mathtt{pid}(p) = \iota \qquad \iota \neq \iota' \qquad \iota' \notin \mathtt{pids}(\Pi)}{p \parallel \Pi \xrightarrow[\iota'\,!\,v]{\iota}\!\!\!\!\rightarrow p' \parallel \Pi}\ \Pi\textsc{Send3}$$

Figure 5.3: Labelled concurrent small-step semantics for CoErl

as long as they do not have side effects. The ΠReceive behaves similarly, except that it allows processes to receive messages from their own mailboxes via the $?\,v$ label. The last three rules are all variations of a process in the system taking a send transition $\iota\,!\,v$, for which there are three possible scenarios:

1. A process ι sends the message v to a different process ι', which exists. This causes the message to be placed at the end of the latter's mailbox (rule ΠSend1).

2. A process ι sends the message v to itself, causing the message to be placed in its own mailbox (rule ΠSend2).

3. A process ι sends a message to ι', where process ι' does not exist, causing the message to be discarded (rule ΠSend3).

5.2.1 Equivalence to the Operational Semantics

The labelled small-step semantics are a transliteration of the unlabelled small-step semantics from section 4.5. As such, the labelled semantics is sound and complete with respect to the unlabelled semantics:

Theorem 5.2.1 (Completeness of the labelled small-step semantics)**.** *The labelled small-step semantics for a single process are complete with respect to the unlabelled small-step semantics (section 4.5):*

$$\forall (x, y) \in (S \times S).\, x \to y \implies \exists \alpha \in A.\, x \xrightarrow{\alpha} y$$

Proof. By induction on the unlabelled small-step relation $\rightarrow$ (figure 4.13). The induction has a case for each rule of the unlabelled small-step semantics, assuming that the premises of the rule hold in each case. We will look at the significant cases (cases which involve communication) and then summarise the remaining cases using an example.

- Rule RECEIVE: Assume values for x and y such that $x \rightarrow y$ holds via rule RECEIVE:

$$\forall cs, e, \rho, \rho_e, \sigma, \iota, mb, mb'.\ mbreceive(cs, mb, \rho) = \mathbf{ok}\ (e, \rho_e, mb') \implies$$
$$(\mathsf{receive}\ cs\ \mathsf{end}, \rho, \sigma, mb, \iota) \rightarrow (e, \rho_e, \sigma, mb', \iota)$$

  Now rewrite the theorem using these assumptions:

$$\forall cs, e, \rho, \rho_e, \sigma, \iota, mb, mb'.\ mbreceive(cs, mb, \rho) = \mathbf{ok}\ (e, \rho_e, mb') \implies$$
$$(\mathsf{receive}\ cs\ \mathsf{end}, \rho, \sigma, mb, \iota) \rightarrow (e, \rho_e, \sigma, mb', \iota) \implies$$
$$\exists \alpha \in A.\ (\mathsf{receive}\ cs\ \mathsf{end}, \rho, \sigma, mb, \iota) \xrightarrow{\alpha} (e, \rho_e, \sigma, mb', \iota)$$

  Next, assume that $\exists v.\ \alpha =?\ v$:

$$\forall cs, e, \rho, \rho_e, \sigma, \iota, mb, mb'.\ mbreceive(cs, mb, \rho) = \mathbf{ok}\ (e, \rho_e, mb') \implies$$
$$(\mathsf{receive}\ cs\ \mathsf{end}, \rho, \sigma, mb, \iota) \rightarrow (e, \rho_e, \sigma, mb', \iota) \implies$$
$$\exists v.\ (\mathsf{receive}\ cs\ \mathsf{end}, \rho, \sigma, mb, \iota) \xrightarrow{?\ v} (e, \rho_e, \sigma, mb', \iota)$$

  This holds by rule ARECEIVE when $\exists v.\ mb' = mbremove(v, mb)$, which is implied by our assumption that $mbreceive(cs, mb, \rho) = \mathbf{ok}\ (e, \rho_e, mb')$ (figure 4.11).

- Rule SEND: Assume values for x and y such that $x \rightarrow y$ holds via rule SEND:

$$\forall v, \rho, \iota', \rho', \sigma, mb, \iota.\ (val\ v, \rho, (\iota'\ !\ \square, \rho') :: \sigma, mb, \iota) \rightarrow (val\ v, \rho', \sigma, mb, \iota) \implies$$
$$\exists \alpha \in A.\ (val\ v, \rho, (\iota'\ !\ \square, \rho') :: \sigma, mb, \iota) \xrightarrow{\alpha} (val\ v, \rho', \sigma, mb, \iota)$$

  This holds by rule ASEND if we assume $\alpha = \iota'\ !\ v$.

- All other rules: trivial via $\alpha = \tau$. For example, in rule NIL: Assume values for x and y such that $x \rightarrow y$ holds via rule NIL:

$$\forall \rho, \sigma, mb, \iota.\ ([\,], \rho, \sigma, mb, \iota) \rightarrow (val\ [\,], \rho, \sigma, mb, \iota) \implies$$
$$\exists \alpha \in A.\ ([\,], \rho, \sigma, mb, \iota) \xrightarrow{\alpha} (val\ [\,], \rho, \sigma, mb, \iota)$$

  This holds by rule ANIL when $\alpha = \tau$. Similarly for all remaining cases.

□

The unlabelled small-step semantics (figure 4.13 on page 61) are deterministic due to the absence of message arrivals. For any process configuration there is at most one rule which can apply. If we permitted message arrivals however, the unlabelled small-step semantics would be non-deterministic. Therefore, we can prove soundness *up to* the arrival of messages, which is sufficient for subsequent theorems:

Theorem 5.2.2 (Soundness of the labelled small-step semantics)**.** *The labelled small-step semantics for a single process are sound with respect to the unlabelled small-step semantics up to the arrival of messages:*

$$\forall (x, \alpha, y) \in (S \times A \times S).\, \nexists m.\, \alpha = \mathrm{arr}\ m \implies x \xrightarrow{\alpha} y \implies x \rightarrow y$$

Proof. By case analysis on α, followed by induction on $x \xrightarrow{\alpha} y$. The induction has a case for each rule of the labelled small-step semantics, assuming that the premises of the rule hold in each case. For the labels $\iota\ !\ m$ and $?\ m$, we know that only one rule of the labelled relation can apply (ASend and AReceive respectively). We consider these two cases in detail and then summarise all remaining cases (where $\alpha = \tau$) using an example.

- Assume α is a receive label, i.e. $\forall v.\, \alpha =?\ v$:

  $$\forall x, y, v.\, x \xrightarrow{?\ v} y \implies x \rightarrow y$$

  By examining the labelled small-step relation, we know that only rule AReceive can apply. Choose values for x and y such that the rule still applies:

  $$\begin{aligned}&\forall cs, e, \rho, \sigma, \iota, mb, mb', v.\\ &\qquad mbreceive(cs, mb, \rho) = \mathbf{ok}\ (e, \rho_e, mb') \implies mb' = mbremove(v, mb) \implies\\ &\qquad\qquad (\texttt{receive cs ende}, \rho, \sigma, mb, \iota) \xrightarrow{?\ v} (e, \rho_e, \sigma, mb', \iota) \implies\\ &\qquad\qquad\qquad (\texttt{receive cs ende}, \rho, \sigma, mb, \iota) \rightarrow (e, \rho_e, \sigma, mb', \iota)\end{aligned}$$

  This holds via rule Receive iff $mbreceive(cs, mb, \rho) = \mathbf{ok}\ (e, \rho_e, mb')$, which is assumed.

- Assume that α is a send label, i.e. $\forall \iota', v.\, \alpha = \iota'\ !\ v$. By examining the labelled small-step relation, we know that only rule ASend can apply. Choose values for x and y such that the rule still applies:

  $$\begin{aligned}&\forall v, \rho, \iota', \rho', \sigma, mb, \iota.\, (val\, v, \rho, (\iota'\ !\ \square, \rho') :: \sigma, mb, \iota) \xrightarrow{\iota'\ !\ v} (val\, v, \rho', \sigma, mb, \iota) \implies\\ &\qquad\qquad (val\, v, \rho, (\iota'\ !\ \square, \rho') :: \sigma, mb, \iota) \rightarrow (val\, v, \rho', \sigma, mb, \iota)\end{aligned}$$

This holds via rule Send.

- $\alpha = \tau$ $x \xrightarrow{\tau} y$ holds for all rules except AReceive, ASend, and AArrive. For example, in rule ANil: We rewrite the theorem by substituting x and y with configurations from the conclusion of rule ANil:

$$\forall \rho, \sigma, mb, \iota.\, ([\,], \rho, \sigma, mb, \iota) \xrightarrow{\tau} (\mathtt{val}\ [\,], \rho, \sigma, mb, \iota) \implies ([\,], \rho, \sigma, mb, \iota) \rightarrow (\mathtt{val}\ [\,], \rho, \sigma, mb, \iota)$$

The relation $x \rightarrow y$ holds via rule Nil. Similarly for all remaining cases.

□

5.2.2 Reflexive Transitive Closure

The reflexive transitive closure of the concurrent semantics represents multiple execution steps of a concurrent system, possibly in a non-deterministic way. In the concurrent semantics, traces take on a slightly different form, as each transition is additionally labelled with the PID of the process which made the transition. As such, traces will be defined as a sequence of tuples:

Definition 5.2.3 (Concurrent Traces)**.** *A concurrent trace* T^{Π} *is either empty* (*via the neutral element* ϵ) *or a label and PID pair* (α, ι) *composed with another trace:*

$$T^{\Pi} ::= \epsilon \mid (\alpha, \iota).T^{\Pi}$$

Traces of a concurrent system represent the events of the *entire* system, i.e. all transitions taken by all processes in the system. Therefore, a trace of the form represents behaviour of processes in a concurrent system over time:

$$(\alpha_1, \iota_1).(\alpha_2, \iota_2).\dots.(\alpha_n, \iota_n).\epsilon$$

There is no requirement that each action or each PID is different, though. The above trace does not imply that each PID $\iota_1, \iota_2, \dots, \iota_n$ is different, nor that they are all the same. Traces can also be viewed as repeated applications of the labelled concurrent small-step relation:

$$s_1 \overset{\iota_1}{\underset{\alpha_1}{\twoheadrightarrow}} s_2, s_2 \overset{\iota_2}{\underset{\alpha_2}{\twoheadrightarrow}} s_3, \dots, s_n \overset{\iota_n}{\underset{\alpha_n}{\twoheadrightarrow}} s_{n+1}$$

$$\dfrac{}{p \overset{\epsilon}{\twoheadrightarrow}{}^{*} p}\ \textsc{Refl} \qquad\qquad \dfrac{p \overset{\alpha}{\underset{\iota}{\twoheadrightarrow}} p' \qquad p' \overset{T}{\twoheadrightarrow}{}^{*} p''}{p \overset{(\alpha,\iota).T}{\twoheadrightarrow}{}^{*} p''}\ \textsc{Trans}$$

Figure 5.4: Reflexive transitive closure of the labelled concurrent semantics

The reflexive transitive closure of the labelled concurrent semantics (figure 5.4) can be also be used to trace behaviour of a concurrent system over multiple steps:

$$\dfrac{s_1 \overset{\iota_1}{\underset{\alpha_1}{\twoheadrightarrow}} s_2 \qquad \dfrac{s_2 \overset{\iota_2}{\underset{\alpha_2}{\twoheadrightarrow}} s_3 \qquad \begin{array}{c} \dfrac{s_n \overset{\iota_n}{\underset{\alpha_n}{\twoheadrightarrow}} s_{n+1} \qquad \dfrac{}{s_{n+1} \overset{\epsilon}{\twoheadrightarrow}{}^{*} s_{n+1}}\ \textsc{Refl}}{s_n \overset{(\alpha_n,\iota_n).\epsilon}{\twoheadrightarrow}{}^{*} s_{n+1}}\ \textsc{Trans} \\ \vdots \end{array}}{s_2 \overset{(\alpha_2,\iota_2).....(\alpha_n,\iota_n).\epsilon}{\twoheadrightarrow}{}^{*} s_{n+1}}\ \textsc{Trans}}{s_1 \overset{(\alpha_1,\iota_1).(\alpha_2,\iota_2).....(\alpha_n,\iota_n).\epsilon}{\twoheadrightarrow}{}^{*} s_{n+1}}\ \textsc{Trans}$$

5.2.3 Non-determinism

The rules of the small-step semantics are non-deterministic, hence the reflexive transitive closure is non-deterministic. While each individual process is deterministic with respect to its mailbox state and the expression being evaluated, the arrival of messages alters the state of the mailbox, possibly altering the branching behaviour of receive expressions in the body of that process. In terms of concurrent systems containing multiple process, the order in which events occur is rather loosely constrained: while sending messages requires some level of synchronisation between the processes involved in the communication, all other execution may be interleaved in a non-deterministic way. As such, the traces obtained by using the reflexive transitive closures may represent one of many possible executions, rather than the only possible execution. There may be other executions of a system which yield identical traces, but a single trace obtained from one execution does not necessarily represent all possible behaviours of the system.

5.3 Trace Equivalence

As of yet, there is no way to compare traces with each other: traces can be obtained from the small-step semantics, but there is no way to check whether two or more traces represent the same behaviour. There are many well-known techniques for analysing so-called "observational" traces of a system (Milner 1980; Nicola 1987; Hoare 1985).

$$\frac{}{\epsilon \overset{T}{=} \epsilon}\ \epsilon\textsc{Strong} \qquad\qquad \frac{\alpha = \alpha' \qquad T \overset{T}{=} T'}{\alpha.T \overset{T}{=} \alpha'.T'}\ \alpha\textsc{Strong}$$

Figure 5.5: Strong trace equivalence

The most straightforward way to compare two traces is to check whether they are identical: they must have the same length, and both traces must have the same elements in the same order. This relationship between two traces is called *strong equivalence* and is defined in figure 5.5. The base case for the relation is the ϵStrong rule, which states that two empty traces are strongly equivalent. The recursive vase is the αStrong rule which requires that both traces are not empty, that the heads of each trace are identical, and that the rest of the trace is strongly equivalent.

This definition, however, requires that the internal labels of both traces are identical. For example, the following two traces are strongly equivalent ($T_1 \overset{T}{=} T_2$):

$$T_1 ::= (\mathtt{arr}\ v).(?\ v).\epsilon$$

$$T_2 ::= (\mathtt{arr}\ v).(?\ v).\epsilon$$

but the following two are not:

$$T_3 ::= (\mathtt{arr}\ v).(?\ v).\epsilon$$

$$T_4 ::= (\mathtt{arr}\ v).(\tau).(?\ v).\epsilon$$

Recall that in section 5.1.1 on page 68, τ was introduced to represent "internal" behaviour of a process, i.e. a transition which does represent sending, receiving, or arriving of messages. Therefore, the definition of strong equivalence requires that *all* behaviour is identical, rather than just the communicating behaviour.

A weaker version of the trace equivalence relation can be used to "step over" these internal transitions, allowing for two traces to be defined as equivalent based solely on the non-τ labels. Figure 5.6 on the following page defines a weaker $\overset{T}{\approx}$ relation, which extends the definition of strong equivalence from figure 5.5 with two more rules. The rule ϵWeak is similar to ϵStrong and αWeak is similar to αStrong. The other two rules τLeft and τRight weaken the relation by allowing a τ label to be ignored in either the left or right traces. No other labels can be ignored, requiring that both traces must have the same non-τ elements in the same order.

This weaker definition of trace equivalence holds for T_3 and T_4 ($T_3 \overset{T}{\approx} T_4$):

$$T_3 ::= (\mathtt{arr}\ v).(?\ v).\epsilon$$

$$T_4 ::= (\mathtt{arr}\ v).(\tau).(?\ v).\epsilon$$

$$\frac{}{\epsilon \overset{T}{\approx} \epsilon}\ \epsilon\textsc{Weak} \qquad \frac{\alpha = \alpha' \qquad T \overset{T}{\approx} T'}{\alpha.T \overset{T}{\approx} \alpha'.T'}\ \alpha\textsc{Weak} \qquad \frac{T \overset{T}{\approx} T'}{\tau.T \overset{T}{\approx} T'}\ \tau\textsc{Left} \qquad \frac{T \overset{T}{\approx} T'}{T \overset{T}{\approx} \tau.T'}\ \tau\textsc{Right}$$

Figure 5.6: Weak trace equivalence

Finally, it can be shown that all strongly equivalent traces are also weakly equivalent:

Theorem 5.3.1 (All strongly-equivalent traces are also weakly-equivalent)**.**

$$\forall T_1, T_2 \in T.\, T_1 \overset{T}{=} T_2 \Longrightarrow T_1 \overset{T}{\approx} T_2$$

Proof. By induction on $T_1 \overset{T}{=} T_2$:

- Rule ϵStrong: The rule only holds when $T_1 = \epsilon$ and $T_2 = \epsilon$, hence we rewrite the theorem accordingly:

$$\epsilon \overset{T}{=} \epsilon \Longrightarrow \epsilon \overset{T}{\approx} \epsilon$$

  The conclusion $\epsilon \overset{T}{\approx} \epsilon$ holds by rule ϵWeak.

- Rule αStrong: The rule only holds when both T_1 has the form $\alpha.T_1'$ and T_2 has the form $\alpha'.T_2$. Therefore we can rewrite the theorem:

$$\forall T_1', T_2', \alpha, \alpha'.\, \alpha.T_1 \overset{T}{=} \alpha'.T_2 \Longrightarrow \alpha.T_1 \overset{T}{\approx} \alpha'.T_2$$

  Assume that $\alpha.T_1 \overset{T}{=} \alpha'.T_2$ by rule αStrong, hence $\alpha = \alpha'$ and $T_1' \overset{T}{=} T_2'$. Induction hypothesis: $T_1 \overset{T}{=} T_2 \Longrightarrow T_1 \overset{T}{\approx} T_2$. Therefore, $\alpha.T_1 \overset{T}{\approx} \alpha'.T_2$ holds by rule αWeak.

□

but that not all weakly equivalent traces are strongly equivalent:

Theorem 5.3.2 (Not all weakly-equivalent traces are also strongly-equivalent)**.**

$$\exists T_1, T_2 \in T.\, T_1 \overset{T}{\approx} T_2 \Longrightarrow T_1 \overset{T}{\neq} T_2$$

Proof. Assume that $T_1 = \epsilon$ and $T_2 = \tau.\epsilon$. Then $\epsilon \overset{T}{\approx} \tau.\epsilon$ by rule τRight. But $\epsilon \overset{T}{=} \tau.\epsilon$ does not hold by rule ϵStrong or αStrong. □

$$\begin{tikzcd}
 & s \arrow[dl, "\alpha_1"'] \arrow[dr, "\alpha_1"] & \\
s_2 \arrow[d, "\alpha_2"'] & & s_3 \arrow[d, "\alpha_3"] \\
s_2' & & s_3'
\end{tikzcd}$$

(a) Branching before α_1

$$\begin{tikzcd}
 & s \arrow[d, "\alpha_1"'] & \\
 & s' \arrow[dl, "\alpha_2"'] \arrow[dr, "\alpha_3"] & \\
s_2'' & & s_3''
\end{tikzcd}$$

(b) Branching after α_1

Figure 5.7: Traces of branching states

5.3.1 Branching Behaviour

One weakness of trace-based analyses is that they cannot be used to determine the exact behaviour of systems which have multiple execution branches. While traces can capture that a system has *multiple* behaviours, they cannot be used to determine with certainty exactly where a system branches during execution (Milner 1980; Roscoe 1998; Hennessy and Milner 1985).

For example, consider the two state machines in figure 5.7: the state machine on the left (figure 5.7a) can branch from the initial state, while the state machine on the right (figure 5.7b) branches after one transition. The state machine in figure 5.7a has two possible traces from an initial state to a final state: $\alpha_1.\alpha_2.\epsilon$ by transitioning via state s_2, and $\alpha_1.\alpha_3.\epsilon$ by transitioning via state s_3. Similarly, there are two possible traces for the state machine in figure 5.7b: $\alpha_1.\alpha_2.\epsilon$ by following the left-hand path to s_2'', and $\alpha_1.\alpha_3.\epsilon$. Although the systems branch at different points, the transitions are labelled in such a way that the set of all execution traces for both systems is identical. It is not always the case that LTSs are labelled so that the branch point cannot be determined, but it is important to note that it is always a possibility that a system *is* labelled in such a way, and there is no way to tell whether or not a system is labelled so that branch points can't be found.

In other words, if it is possible to construct more than one relation and accompanying set of states which produce the same set of traces, then it is impossible to uniquely determine which relation and which set of states was originally used to construct the system.

Regardless, traces are a useful way to analyse and compare the behaviour of communicating CoErl systems, though its limitations must be recognised and taken into account. Specifically, we should be aware that although multiple programs may exhibit identical communicating behaviour, they do not necessarily have the same control flow diagrams.

5.4 Re-Ordering and Insertion of Arrivals

The non-deterministic behaviour of a CoErl process comes from its mailbox: as the process executes, new messages may arrive in its mailbox, changing how receive expressions are evaluated. Looking back at chapter 4 on page 43, the execution of a process is entirely syntax-directed, except when messages arrive *during* execution. Therefore, assuming that no messages arrive during execution, the evaluation of a process is deterministic:

Theorem 5.4.1 (The execution of a process is deterministic in the absence of any arrival events)**.** *For all executions of a process* p *to a final state* p_1 *or* p_2 (*i.e. such that* $\nexists p_1'.\, p_1 \to p_1'$ *and* $\nexists p_2'.\, p_2 \to p_2'$), *and assuming that* t_1 *and* t_2 *are traces which do not contain any arrive events* ($\forall m.\, \mathtt{arr}\ m \notin t_1$ *and* $\forall m.\, \mathtt{arr}\ m \notin t_2$), *then* p_1 *and* p_2 *are the same:*

$$p \xrightarrow{t_1}{}^{*} p_1 \implies p \xrightarrow{t_2}{}^{*} p_2 \implies p_1 = p_2$$

Proof.

- Show that for all p, p', p'', $\alpha.\, \nexists v.\, \alpha = \mathtt{arr}\ v$, $\alpha'.\, \nexists v.\, \alpha' = \mathtt{arr}\ v$, that where $p \xrightarrow{\alpha} p'$ and $p \xrightarrow{\alpha'} p''$, then $p' = p''$ and $\alpha = \alpha'$. For each configuration p there is only one rule which applies, hence $p' = p''$ and $\alpha = \alpha'$. The rule AArrive can never apply due to the restrictions on the value of α and α'.
- Show that $\xrightarrow{T}{}^{*}$ is deterministic via induction on the relation, using proof that $\xrightarrow{\alpha}$ is determinstic when $\nexists v.\, \alpha = \mathtt{arr}\ v$.

□

This observation is not very useful on its own: a system without communication during execution is not very representative of Erlang's communication model. A more interesting observation is that the precise timing of message arrivals is not always important in determining the execution of a process. As communication in CoErl is asynchronous, messages can arrive before they are required. For example, in Erlang, the two programs in listing 9 on the next page produce the same output.

Both programs have the same client code: the function `client/0` first receives the message `'ready'` which indicates the start of the test. Afterwards, the client calls the `print_mailbox/0` function which uses debugging functions to directly inspect the state of the mailbox, printing

```
-module(no_preload).
-export([start/0]).

print_mailbox() ->
  {_, Msgs} =
   erlang:process_info(self(),messages),
  io:format("My mailbox: ~p~n", [Msgs]).

client() ->
  receive ready ->
    io:format("Ready~n")
  end,
  print_mailbox(),
  receive Msg ->
    io:format("1: ~p~n", [Msg])
  end,
  print_mailbox(),
  receive Msg2 ->
   io:format("2: ~p~n", [Msg2])
  end,
  print_mailbox().

start() ->
  Pid = spawn(fun client/0),
  Pid ! ready,
  timer:sleep(1000),
  Pid ! one,
  timer:sleep(1000),
  Pid ! two,
  ok.
```

(a) Interleaving of arrivals and receives

```
-module(preload).
-export([start/0]).

print_mailbox() ->
  {_, Msgs} =
   erlang:process_info(self(),messages),
  io:format("My mailbox: ~p~n", [Msgs]).

client() ->
  receive ready ->
    io:format("Ready~n")
  end,
  print_mailbox(),
  receive Msg ->
    io:format("1: ~p~n", [Msg])
  end,
  print_mailbox(),
  receive Msg2 ->
    io:format("2: ~p~n", [Msg2])
  end,
  print_mailbox().

start() ->
  Pid = spawn(fun client/0),
  erlang:suspend_process(Pid),
  Pid ! ready,
  Pid ! one,
  Pid ! two,
  erlang:resume_process(Pid),
  ok.
```

(b) Messages arriving before execution

Listing 9: Messages arriving at different times producing the same output

it to the shell. The client then receives a message and prints it, checks its own mailbox again, receives another message, and then prints its final mailbox state.

In listing 9a, the sending `start/0` process first spawns the client, sends the `'ready'` message, then waits before sending the message `'one'`, waits again, and sends the message `'two'`. The use of `timer:sleep/1` ensures with a reasonable level of confidence that all messages arrive at and are received by the client process in a serial fashion. When this program is run in an Erlang shell, the following output is always seen[3]:

```
$ erl

1> no_preload:start().
```

[3]barring high system loads which would affect the scheduling of the client process

```
Ready
My mailbox: []
1: one
My mailbox: []
2: two
My mailbox: []
2> % return value omitted
```

From the perspective of the client process, the following events occur:

1. The `'ready'` message arrives.
2. The `'ready'` message is received.
3. The `'one'` message arrives.
4. The `'one'` message is received.
5. The `'two'` message arrives.
6. The `'two'` message is received.

On the other hand, the sending process in listing 9b on the previous page behaves very differently: it "preloads" the mailbox before allowing the client process to execute. In order to prevent the client process from receiving any messages, it uses the debugging function `erlang:suspend_process/1` to instruct the BEAM to prevent the client process from being scheduled for execution. Then, the sending process sends the `'ready'`, `'one'`, and `'two'` messages before instructing the BEAM to resume the client process so it can be scheduled once again. Running the `start/0` function for this example yields the following output:

```
$ erl
1> preload:start().
Ready
My mailbox: [one,two]
1: one
My mailbox: [two]
2: two
My mailbox: []
2> % return value omitted
```

This client observes the following events:

1. The `'ready'` message arrives.
2. The `'one'` message arrives.
3. The `'two'` message arrives.
4. The `'ready'` message is received.
5. The `'one'` message is received.
6. The `'two'` message is received.

The first example (listing 9a on page 84) interleaves message arrivals with receives and ends with an empty mailbox; the second example (listing 9b on page 84) has all arrivals occur before the receives. Nonetheless, the evaluation of `client/0` was identical in both cases, and both client processes ended with an empty mailbox.

The key observation here is that while the order in which messages arrive has *not* been changed between examples, they have arrived *earlier* in the second example. In CoErl, this equates to the following theorem:

Theorem 5.4.2 (Reordering of arrivals)**.** *For all labels α which are not arrivals ($\forall m.\ \alpha \neq \mathsf{arr}\ m$), if an arrival is observed to occur after α, then it is possible for the arrival to occur before α without affecting execution:*

$$p \xrightarrow{\alpha} p' \xrightarrow{\mathsf{arr}\ m} p''' \implies p \xrightarrow{\mathsf{arr}\ m} p'' \xrightarrow{\alpha} p'''$$

Proof. First, assume that $p = (e, \rho, \sigma, \iota, mb)$:

$$(e, \rho, \sigma, \iota, mb) \xrightarrow{\alpha} p' \xrightarrow{\mathsf{arr}\ m} p''' \implies (e, \rho, \sigma, \iota, mb) \xrightarrow{\mathsf{arr}\ m} p'' \xrightarrow{\alpha} p'''$$

By rule AArrive we can determine the value of p'':

$$(e, \rho, \sigma, \iota, mb) \xrightarrow{\alpha} p' \xrightarrow{\mathsf{arr}\ m} p''' \implies (e, \rho, \sigma, \iota, mb) \xrightarrow{\mathsf{arr}\ m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\alpha} p'''$$

Also assume that $p' = (e', \rho', \sigma', \iota, mb')$ and repeat the process to determine p''':

$$(e, \rho, \sigma, \iota, mb) \xrightarrow{\alpha} (e', \rho', \sigma', \iota, mb') \xrightarrow{\mathsf{arr}\ m} (e', \rho', \sigma', \iota, mb' \mathbin{++} v) \implies$$
$$(e, \rho, \sigma, \iota, mb) \xrightarrow{\mathsf{arr}\ m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\alpha} (e''', \rho''', \sigma''', \iota, mb''')$$

Now, we perform case analysis on α:

- Assume $\alpha = \tau$. First, we note that for any process configuration, internal transitions are determinstic *regardless of the mailbox state*:

$$\begin{aligned}&\forall e, e', \rho, \rho', \sigma, \sigma', \iota, mb_1, mb_2.\\ &\qquad (e, \rho, \sigma, \iota, mb_1) \xrightarrow{\tau} (e', \rho', \sigma', \iota, mb_1) \iff (e, \rho, \sigma, \iota, mb_2) \xrightarrow{\tau} (e', \rho', \sigma', \iota, mb_2) \qquad (5.1)\end{aligned}$$

  This holds by case analysis of the labelled small-step relation. Next, we restate the theorem using $\alpha = \tau$:

$$\begin{aligned}&(e, \rho, \sigma, \iota, mb) \xrightarrow{\tau} (e', \rho', \sigma', \iota, mb') \xrightarrow{\text{arr } m} (e', \rho', \sigma', \iota, mb' \mathbin{++} v) \implies\\ &\qquad (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\tau} (e''', \rho''', \sigma''', \iota, mb''')\end{aligned}$$

  By case analysis on $\xrightarrow{\tau}$ we also know that $e''' = e'$, $\rho''' = \rho'$, and $\sigma''' = \sigma'$.

$$\begin{aligned}&(e, \rho, \sigma, \iota, mb) \xrightarrow{\tau} (e', \rho', \sigma', \iota, mb') \xrightarrow{\text{arr } m} (e', \rho', \sigma', \iota, mb' \mathbin{++} v) \implies\\ &\qquad (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\tau} (e', \rho', \sigma', \iota, mb''')\end{aligned}$$

  By equation (5.1), we also know that $mb' = mb$ and $mb''' = mb \mathbin{++} v$:

$$\begin{aligned}&(e, \rho, \sigma, \iota, mb) \xrightarrow{\tau} (e', \rho', \sigma', \iota, mb) \xrightarrow{\text{arr } m} (e', \rho', \sigma', \iota, mb \mathbin{++} v) \implies\\ &\qquad (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\tau} (e', \rho', \sigma', \iota, mb \mathbin{++} v) \qquad (5.2)\end{aligned}$$

  which holds via the definition of the labelled small-step relation.

- Assume $\forall \iota', m',. \alpha = \iota' \mathbin{!} m'$. Restate the theorem using $\alpha = \iota' \mathbin{!} m'$:

$$\begin{aligned}&(e, \rho, \sigma, \iota, mb) \xrightarrow{\iota' \,!\, m'} (e', \rho', \sigma', \iota, mb') \xrightarrow{\text{arr } m} (e', \rho', \sigma', \iota, mb' \mathbin{++} v) \implies\\ &\qquad (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\iota' \,!\, m'} (e''', \rho''', \sigma''', \iota, mb''')\end{aligned}$$

  Only rule ASend may apply when $\alpha = \iota' \mathbin{!} m'$, so we can determine that $e' = e$, $\rho' = \rho$, $\sigma' = \sigma$, and $mb' = mb$:

$$\begin{aligned}&(e, \rho, \sigma, \iota, mb) \xrightarrow{\iota' \,!\, m'} (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \implies\\ &\qquad (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\iota' \,!\, m'} (e''', \rho''', \sigma''', \iota, mb''')\end{aligned}$$

  We repeat the process again for e''', ρ''', σ''', and mb''':

$$\begin{aligned}&(e, \rho, \sigma, \iota, mb) \xrightarrow{\iota' \,!\, m'} (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \implies\\ &\qquad (e, \rho, \sigma, \iota, mb) \xrightarrow{\text{arr } m} (e, \rho, \sigma, \iota, mb \mathbin{++} v) \xrightarrow{\iota' \,!\, m'} (e, \rho, \sigma, \iota, mb \mathbin{++} v)\end{aligned}$$

  which holds by the definition of the labelled small-step relation.

$$\begin{tikzcd}
 & p \arrow[dl, "\alpha"'] \arrow[dr, "\mathsf{arr}\ m"] & \\
p' \arrow[dr, "\mathsf{arr}\ m"'] & & p'' \arrow[dl, "\alpha"] \\
 & p''' &
\end{tikzcd}$$

where $\forall m.\ \alpha \neq \mathsf{arr}\ m$

Figure 5.8: Commutative properties of arrival events

- Assume $\forall m'.\ \alpha =?\ m'$. Restate the theorem using $\alpha =?\ m'$:

$$(e, \rho, \sigma, \iota, mb) \xrightarrow{?\ m'} (e', \rho', \sigma', \iota, mb') \xrightarrow{\mathsf{arr}\ m} (e', \rho', \sigma', \iota, mb' +\!\!+\, v) \implies$$
$$(e, \rho, \sigma, \iota, mb) \xrightarrow{\mathsf{arr}\ m} (e, \rho, \sigma, \iota, mb +\!\!+\, v) \xrightarrow{?\ m'} (e''', \rho''', \sigma''', \iota, mb''')$$

Only rule AReceive may apply when $\alpha =?\ m'$, so we can determine that $\sigma' = \sigma$, $\sigma''' = \sigma$, $mb' = mbremove(m', mb)$, and $mb''' = mbremove(m', mb +\!\!+\, m)$:

$$(e, \rho, \sigma, \iota, mb) \xrightarrow{?\ m'} (e', \rho', \sigma, \iota, mbremove(m', mb)) \xrightarrow{\mathsf{arr}\ m}$$
$$(e', \rho', \sigma, \iota, mbremove(m', mb) +\!\!+\, m) \implies$$
$$(e, \rho, \sigma, \iota, mb) \xrightarrow{\mathsf{arr}\ m} (e, \rho, \sigma, \iota, mb +\!\!+\, v) \xrightarrow{?\ m'}$$
$$(e''', \rho''', \sigma, \iota, mbremove(m', mb +\!\!+\, m))$$

Finally, we can prove $p'' = p''$ by showing that $e''' = e'$, $\rho''' = \rho$, and also:

$$mbremove(m', mb') +\!\!+\, m = mbremove(m', mb +\!\!+\, m)$$

This is possible by proving the following property of $mbreceive$:

$$\forall cs, e, \rho, \rho', e, mb, mb'.\ mbreceive(cs, mb, \rho) = \mathbf{ok}\ ((e, \rho'), mb') \implies$$
$$mbreceive(cs, mb +\!\!+\, m, \rho) = \mathbf{ok}\ ((e, \rho'), mb' +\!\!+\, m)$$

which is true by induction on the definitions of $mbreceive$, $mbselect$, and $mbremove$. Hence $p'' = p'''$.

□

The diagram in figure 5.8 shows how such a re-ordering of events is commutative: if it's possible to perform a non-arrival action before the arrival of a message, then it's also possible to perform the arrival first and end up in the same state.

Note that both the preceding theorem and the diagram in figure 5.8 on the preceding page require that the "other event" is *not* an arrival: if the order of arrivals changed, so would the order of messages in the mailbox. If the order of messages in the mailbox changed, the result of receive expressions could change depending on the clauses in them. For example, if we changed the order of the second and third messages sent in either of the programs from listing 9 on page 84 (i.e if we swapped `Pid ! one` and `Pid ! two`), we would observe the message `'two'` being received before `'one'`. In the case of listing 9a on page 84, swapping the order of the second and third sent messages yields a *different* output:

```
$ erl
1> no_preload:start().
Ready
My mailbox: []
1: one
My mailbox: []
2: two
My mailbox: []
2> % return value omitted
```

Since any arrive event can occur one step earlier (theorem 5.4.2), it can also be shown that any arrive in a *trace* can arrive one step earlier:

Theorem 5.4.3. *For all traces* t *and non-arrive labels* α $(\forall m.\, \alpha \neq \mathrm{arr}\ m)$*, an arrival can occur one step earlier:*

$$\forall p, p' \in S.\, p \xrightarrow{\alpha.\mathrm{arr}\ m.t}{}^{*} p' \implies p \xrightarrow{\mathrm{arr}\ m.\alpha.t}{}^{*} p'$$

Proof. By the transitive rule for $\rightarrow^{*}$, it is known that there exists a p_1 and p_2 such that:

$$p \xrightarrow{\alpha.\mathrm{arr}\ m.t}{}^{*} p' = p \xrightarrow{\alpha} p_1 \xrightarrow{\mathrm{arr}\ m} p_2 \xrightarrow{t}{}^{*} p' \tag{5.3}$$

Therefore, the theorem can be restated as:

$$p \xrightarrow{\alpha} p_1 \xrightarrow{\mathrm{arr}\ m} p_2 \xrightarrow{t}{}^{*} p' \implies p \xrightarrow{\mathrm{arr}\ m.\alpha.t}{}^{*} p' \tag{5.4}$$

By theorem 5.4.2 on page 86, we can show there exists a p_1' such that $p \xrightarrow{\alpha} p_1 \xrightarrow{\text{arr } m} p_2 \implies p \xrightarrow{\text{arr } m} p_1' \xrightarrow{\alpha} p_2$. Therefore, we can rewrite (5.4) to:

$$p \xrightarrow{\text{arr } m} p_1' \xrightarrow{\alpha} p_2 \xrightarrow{t}{}^{*} p' \implies p \xrightarrow{\text{arr } m.\alpha.t}{}^{*} p' \tag{5.5}$$

And applying the transitive rule for $\rightarrow^*$:

$$p \xrightarrow{\text{arr } m.\alpha.t}{}^{*} p' \implies p \xrightarrow{\text{arr } m.\alpha.t}{}^{*} p' \tag{5.6}$$

□

The theorem states that for any individual process, an arrival can take place earlier without affecting the final state of the process. Importantly, the theorem does not make any assertions about concurrent CoErl systems, where such a theorem would show that a message could arrive before it is sent. Instead, the theorem gives a causality between arrivals and receives: each message must arrive strictly before it is received, the message may arrive arbitrarily early as long as the order of arrivals is not changed.

5.5 Trace Replay

As discussed in section 5.1 on page 67, a desired property of the LTS model is to be able to replay a trace on a mailbox, in order to reconstruct a process' final mailbox state given its initial mailbox and a trace of its execution. The desired property was stated as:

$$\forall (p, \alpha, p') \in R.\, \mathrm{apply}(\alpha, \mathrm{mailbox}(p)) = \mathrm{mailbox}(p')$$

where R is the labelled small-step relation ($\xrightarrow{\alpha}$) from section 5.1 on page 67.

Definition 5.5.1 (Trace Application)**.** *Given a mailbox* mb*, the effects of an event* α *can be applied to the mailbox via the* apply *function:*

$$\mathrm{apply}(\alpha, mb) \stackrel{\text{def}}{=} \begin{cases} mb & \textit{if } \alpha = \tau \\ mb & \textit{if } \alpha = \iota \,!\, m \\ mb \mathbin{++} m & \textit{if } \alpha = \text{arr } m \\ \mathrm{remove}(m, mb) & \textit{if } \alpha = ?\, m \end{cases}$$

Theorem 5.5.1.

$$\forall p, p' \in S.\, p \xrightarrow{\alpha} p' \implies \mathrm{apply}(\alpha, \mathrm{mailbox}(p)) = \mathrm{mailbox}(p')$$

Proof. By case analysis of α, there are four possible labels:

- τ – we must prove $\forall p, p' \in S.\, p \xrightarrow{\tau} p' \implies \mathtt{apply}(\tau, \mathtt{mailbox}(p)) = \mathtt{mailbox}(p')$. For all rules of the form $p \xrightarrow{\tau} p'$, the mailbox is not modified, i.e. $\mathtt{mailbox}(p) = \mathtt{mailbox}(p')$. Therefore, the goal can be rewritten as $\mathtt{apply}(\tau, \mathtt{mailbox}(p)) = \mathtt{mailbox}(p)$, which is true by definition of $\mathtt{apply}$.

- $\mathtt{arr}\ m$ – only one rule has the form $p \xrightarrow{\mathtt{arr}\ m} p'$. The arrive rule appends the message m to the mailbox of p, such that $\mathtt{mailbox}(p) +\!\!+\, m = \mathtt{mailbox}(p')$. Therefore, the goal can be rewritten as $\mathtt{apply}(\mathtt{arr}\ m, \mathtt{maibox}(p)) = \mathtt{mailbox}(p) +\!\!+\, m$, which is true by definition of $\mathtt{apply}$.

- $\iota\ !\ m$ – no rule of the form $p \xrightarrow{\iota\ !\ m} p'$ modifies the mailbox of p, so $\mathtt{mailbox}(p) = \mathtt{mailbox}(p')$. Therefore similar to τ case.

- $?\ m$ – only one rule has the form $p \xrightarrow{?\ m} p'$. The receive rule removes the first message m from the mailbox of p, such that $\mathtt{remove}(m, \mathtt{mailbox}(p)) = \mathtt{mailbox}(p')$. Therefore, the goal can be rewritten as $\mathtt{apply}(\mathtt{receive}\ m\ \mathtt{end}, \mathtt{mailbox}(p)) = \mathtt{remove}(m, \mathtt{mailbox}(p))$, which is true by definition of $\mathtt{apply}$.

□

The $\iota\ !\ m$ case of the proof might be surprising, as it might be assumed that p is sending a message to itself (if $\iota = \mathtt{pid}(p)$). However, such cases are handled by the concurrent labelled small-step semantics, where any self-addressed send results in an arrive event immediately afterwards.

The definition can be extended from a single event to an entire trace, enabling the replay of a sequence of events upon a mailbox:

Definition 5.5.2 (Trace Replay)**.** *Given an initial mailbox* mb*, the effects of the trace* T *can be replayed on* mb *by the following function:*

$$\mathtt{replay}(T, mb) \overset{\text{def}}{=} \begin{cases} mb & \textit{if } T = \epsilon \\ \mathtt{replay}(T', \mathtt{apply}(\alpha, mb)) & \textit{if } T = \alpha.T' \end{cases}$$

This comes with an accompanying theorem:

Theorem 5.5.2.

$$\forall p, p' \in S.\, p \xrightarrow{T}{}^{*} p'' \implies \mathtt{replay}(T, \mathtt{mailbox}(p)) = \mathtt{mailbox}(p'')$$

Proof. By induction on $p \xrightarrow{T}{}^{*} p''$.

- Base case. The trace must be empty by the definition of the reflexive rule, i.e. $T = \epsilon$. Also, $p = p''$. Therefore, we must prove $p \xrightarrow{\epsilon}{}^{*} p \implies \mathtt{replay}(\epsilon, \mathtt{mailbox}(p)) = \mathtt{mailbox}(p)$, which is true by definition of $\mathtt{replay}$.

- Inductive case. The trace must be non-empty by definition of the transitive rule, i.e. $T = \alpha.T'$. As per the transitive case, assume that $p \xrightarrow{\alpha} p'$ and $p' \xrightarrow{T'}{}^{*} p''$. We must prove $p \xrightarrow{\alpha.T'}{}^{*} p'' \implies \mathtt{replay}(\alpha.T', \mathtt{mailbox}(p)) = \mathtt{mailbox}(p'')$. By induction, also assume that $p' \xrightarrow{T'}{}^{*} p'' \implies \mathtt{replay}(T', \mathtt{mailbox}(p')) = \mathtt{mailbox}(p'')$. Simplification of $\mathtt{replay}$ yields $p \xrightarrow{\alpha.T}{}^{*} p'' \implies \mathtt{replay}(T', \mathtt{apply}(\alpha, \mathtt{mailbox}(p))) = \mathtt{mailbox}(p'')$. By theorem 5.5.1, we also know that $\mathtt{apply}(\alpha, \mathtt{mailbox}(p)) = \mathtt{mailbox}(p')$. Substitution of $\mathtt{mailbox}(p')$ gives $\mathtt{replay}(T', \mathtt{mailbox}(p')) = \mathtt{mailbox}(p'')$, which is true by assumption.

□

5.6 Orphan Messages in Traces

The LTS model of CoErl allows the communicating behaviour of processes and concurrent systems to be considered separately from the internal behaviour of processes. Insight into the relationship between the arrival and receipt of messages has yielded a normalisation technique for traces, which allows traces to be modulo the precise timing of arrivals. The normalisation technique maintains the order of arrivals in relation to oneanother, but does permit an arrival to appear at earlier positions in a trace.

For the purpose of detecting communication discrepancies in Erlang programs, the theory of traces must be related to the observed behaviour of orphan messages in real Erlang programs. Informally, an orphan message is a message which arrives in a process' mailbox and is never received. In terms of traces, this can be characterised as an arrival event, and the lack of any corresponding receive event following it:

$$\mathtt{simple\text{-}orphan}(m, T) \overset{\text{def}}{=} \begin{cases} \alpha = \mathtt{arr}\ m \wedge ?\ m \notin T' & \text{if } T = \alpha.T' \\ \mathtt{simple\text{-}orphan}(m, T') & \text{if } T = \alpha.T' \\ \mathit{false} & \text{otherwise} \end{cases}$$

For the purposes of detecting communication discrepancies, traces of CoErl programs can be used to formally characterise the notion of an orphan message. With respect to finite traces (i.e. ϵ-terminated traces), orphan messages can be readily detected: any message m which arrives but is never received can be considered an orphan. For example, the message m in the following trace is an orphan:

$$\mathsf{arr}\ m.\mathsf{arr}\ n.?\ n.\epsilon$$

The message m arrived, then the message n arrived, after which n was received. In this case, the message m remained in the mailbox at the end of the trace, hence it is an orphan.

5.7 Infinite Computations

In section 4.6 on page 65 we discussed how the operational semantics of CoErl can be used to model non-terminating computations. The unlabelled small-step operational semantics can be used to create a finite approximation of the execution of a non-terminating process. By using the reflexive transitive closure, discrete computational steps can be chained one after another until the process enters a final state or a finite number of steps have been taken. A similar principle applies to the labelled small-step operational semantics from figure 5.1 on page 71 and on page 72: the reflexive transitive closure over the labelled relation can be used to create a finite trace which approximates the communicating behaviour of a non-terminating process.

Theorem 5.4.3 on page 89 can be used to reason about the order of arrivals for finite traces which approximate the behaviour of infinite computations. Additionally, as theorem 5.4.3 applies *for all* traces with tail t, the theorem can even be used to re-order arrivals in an infinite trace. We cannot, however, use this theory of communication traces to reason about orphan messages in infinite computations because a message is only an orphan if it is *never* received. In order to determine that a message is never received we must have the entire trace available, and it is not evident from the trace whether a control flow path exists in the program which would receive the message at an indeterminate point in the future. It may be sufficient to determine that a particular message is not an orphan by using a finite approximation of a process' behaviour, i.e. up until the point at which that message is received, but this may not always be possible and the number of steps required to receive the message cannot be determined ahead of time.

Chapter 6

A Sub-Typing Relation for CoErl

Erlang is a programming language of two parts: functional programming and concurrency. The functional aspect of the language is driven by the type and structure of data, where programmers typically make control flow decisions using patterns, guards, and a specific order of clauses. On the other hand, the concurrent part of the language uses the principles of process isolation and mailboxes to facilitate highly concurrent applications which can run across multiple machines. Communication is where the functional and concurrent aspects of the language meet: pattern matching and guard evaluation determine which messages are removed from the mailbox, possibly affecting future control flow decisions. In order to facilitate pattern matching and guard evaluation at runtime, the BEAM uses tagged values, with each tag indicating whether a value is an atom, an integer, a cons cell, and so forth.

Despite Erlang's dynamic approach to typing, patterns and guards contain important type information which can be used to statically determine the types of values they will accept. The same reasoning can also be used to reason about the *types* of messages a process will receive by statically analysing the `receive` expressions present in its code. The relationship between multiple receive expressions and even the clauses within an individual receive expression are complex, and this chapter aims to shed light on their behaviour.

This chapter introduces a static analysis technique to detect orphan messages at compile time. To achieve this, Erlang's existing type is extended with *intersection* and *negation* types to allow for more precise modelling of real program behaviour, accompanied by a set-based denotational semantics. The denotational semantics is then used as a base for a definition of sub-typing, which characterises the relationship between types in Erlang. This is followed by a sound and complete sub-typing algorithm for the system. All of this is then used to define a type inference system for CoErl patterns, guards, and clauses which can be used to statically determine the

$$
\begin{aligned}
T^p &::= \mathit{atom}() \mid \mathit{boolean}() \mid \mathit{float}() \mid \mathit{integer}() \mid \mathit{number}() \\
&\mid \mathit{pid}() \mid \mathit{reference}() \mid \mathit{port}() \\
&\mid \mathit{list}() \mid \mathit{tuple}() \\
&\mid \mathit{term}() \\
L^i &::= \text{any Erlang integer} \\
L^a &::= \text{any Erlang atom} \\
L &::= \text{'}L^i\text{'} \mid \text{'}L^a\text{'} \\
S, T &::= T^p \mid L \\
&\mid [\,] \mid [\,T_h \mid T_t\,] \mid \{\,\} \mid \{T_1, T_2, \dots, T_n\} \\
&\mid S \sqcup T \mid S \sqcap T \mid \neg T
\end{aligned}
$$

Figure 6.1: Syntax of Types

types of messages that can be received by a process.

The original work on which this work is based presents proofs of soundness and completeness for the functions DNF and CAN, and also presents a discussion on the time complexity of this algorithm: it is exponential in the worst case due to the first step of converting the type into DNF, which can cause an exponential explosion in the size of the type (Pearce 2013).

For CoErl, the main differences to the original work are the primitive types used in the system, leading to different definitions of positive type intersection (figure 6.4 on page 103) and sub-typing (figure 6.5 on page 104). The only other change over the original work is the introduction of cons cells, which behave similarly to tuples.

6.1 Type Syntax

Erlang already has a type syntax used for documentation purposes and as type annotations for static analysis tools (Erlang/OTP Team 2018, Types and Function Specifications). This syntax supports each of Erlang's existing data types, unions of types, and literal values in types. Furthermore, even though Erlang does not allow the programmer to define any new constructors, the type syntax supports user-defined types, which are simply aliases for existing types.

The syntax of types is given in figure 6.1. The type syntax consists of *primitive types* (T^p) which represent the primitive Erlang data types, *literal* integers and atoms (L^i and L^a), list types ($[\,]$, $[\,T_h \mid T_t\,]$), and tuple types ($\{\,\}$, $\{T_1, T_2, \dots, T_n\}$). Furthermore, the syntax $S \sqcup T$ is used to represent the union of types S and T, the syntax $S \sqcap T$ for the intersection of S and T, and $\neg T$ for the negation of T.

The primitive types (T^p) reflect names of type test BIFs in the Erlang/OTP standard library, where for each type of the form $\mathtt{type}()$, there is a BIF named `is_type/1`. Therefore, each of these types is not necessarily distinct: the type $\mathtt{boolean}()$ is a subset of $\mathtt{atom}()$[1], both $\mathtt{integer}()$ and $\mathtt{float}()$ are subsets of $\mathtt{number}()$, and $\mathtt{term}()$ represents *all* Erlang values.

Literal types (L) represent a nuance of Erlang's type syntax for what are essentially singleton types, i.e. types of a single value. As Erlang does not support user-defined data types, these singletons consist of values for existing data types, specifically atoms (L^a) and integers (L^i). For example, the type '2' represents "all values which are the integer 2", and the type ''foo'' represents "all values which are the atom 'foo'". Singleton types often appear when writing type specifications for functions which deal with user-defined data structures:

```
-module(shapes).
-export([area/1]).

-type shape() :: {'square',number(),number()} | {'circle',number()}.

-spec area(shape()) -> number().
area({'square',X,Y}) -> X * Y;
area({'circle',R}) -> math:pi() * math:pow(R,2).
```

In the example above we see a user-defined shape type which is a union of two types: '{'square', $\mathtt{number}()$, $\mathtt{number}$}' and '{'circle', $\mathtt{number}()$}'. Here, the atoms 'square' and 'circle' are singleton types, i.e. types with a single value.

Lists in Erlang are created using either the empty list constructor $[\,]$ (also called *nil*) or by the cons cell constructor $[\,h \mid t\,]$ which consists of a *head* h and tail t. Unlike some other functional programming languages, lists in Erlang may be improper: the tail of a cons cell does not have to be a list. List types are written using the exact same syntax: $[\,]$ represents the type of empty lists, and $[\,T_h \mid T_t\,]$ represents the type of cons cells with heads of type T_h and tails of type T_t. The type $\mathtt{list}()$ in T^p represents the type of all cons cells and all empty lists.

Tuples are also present in the type syntax: they are either empty ($\{\,\}$) or they have $n > 0$ elements ($\{T_1, T_2, \ldots, T_n\}$), where each element has a type. Similarly to lists, the $\mathtt{tuple}()$ type in T^p is used to represent all tuples of all sizes.

Finally, three type operators are given: union ($\sqcup$), intersection ($\sqcap$), and negation ($\neg$). A union of types $S \sqcup T$ represents all members of type S *and also* all members of type T, while the

[1]as boolean values in Erlang are the atoms 'true' and 'false'

intersection of types $S \sqcap T$ represents all members of S which *are also* members of T. The syntax $\neg T$ represents all Erlang terms which *are not* members of type T.

6.2 Denotational Semantics

While the *syntax* of types has been given, the *meaning* of types is as yet undefined. The meaning of some types was alluded to in the previous section: the $\mathtt{tuple}()$ type contains all tuple values, and the negation of a type T contains all values *not* in the type T. One way of giving meaning is via a denotational semantics: constructing mathematical objects which describe the meaning of types. As types in Erlang always reflect one or more distinct values, we will use sets as the mathematical objects which represent types.

The general concept is to associate every possible type T with a set representing the Erlang values which inhabit the type. Semantic brackets will be used for the denotational semantics of types: $[\![T]\!]$ means "the set of values denoted by type T".

These denotational semantics will serve as the source of truth for any static analysis or typing algorithm: soundness and completeness will be judged relative to the denotational semantics.

The denotational semantics for types is given via the definition of $[\![\cdot]\!]$ in figure 6.2 on the next page.

We start with Erlang's primitive types, defined via the syntax of T^p in figure 6.1. The type $\mathtt{term}()$ denotes the infinite set of *all* Erlang values, written $\mathcal{U}$. Other primitives follow, with $\mathtt{atom}()$, $\mathtt{integer}()$, $\mathtt{float}()$, $\mathtt{pid}()$, $\mathtt{reference}()$, $\mathtt{port}()$, $\mathtt{list}()$, and $\mathtt{tuple}()$ each representing the set of Erlang values which inhabit that type. The exception to the norm is $\mathtt{boolean}()$, which is a finite set consisting of the atoms `'true'` and `'false'`. Literal types (from L in figure 6.1) are denoted by a singleton set, whose member is the literal itself.

Empty lists are represented by the singleton set containing the $[\,]$ constructor. Cons cells of the form $[\,T_h \mid T_t\,]$ denote a set of cons cells whose members are every possible pairing of members from $[\![T_h]\!]$ and $[\![T_t]\!]$, such that if $[\![T_h]\!] = \{a, b\}$ and $[\![T_t]\!] = \{x, y\}$, then $[\![[\,T_h \mid T_t\,]]\!] = \{[\,a \mid x\,], [\,a \mid y\,], [\,b \mid x\,], [\,b \mid y\,]\}$. The denotations for tuples are similar, with the empty tuple $\{\,\}$ denoting a singleton set, and the non-empty tuple $\{T_1, T_2, \ldots, T_n\}$ being represented by element-wise pairings.

Type operators translate directly into set operators: the union $S \sqcup T$ is denoted by the union of the denotations of S and T, the intersection $S \sqcap T$ is denoted by the intersection of the denotations, and the negation of T is denoted by the complement of the denotation of T.

Some theorems about the type operators are available "for free" owing to the direct translation of the type operators into set theory:

Primitive types

$$
\begin{aligned}
[\![\mathtt{term}()]\!] &\overset{\text{def}}{=} \mathcal{U} \\
[\![\mathtt{atom}()]\!] &\overset{\text{def}}{=} \text{the infinite set of all Erlang atoms} \\
[\![\mathtt{boolean}()]\!] &\overset{\text{def}}{=} \{\texttt{'true'},\texttt{'false'}\} \\
[\![\mathtt{integer}()]\!] &\overset{\text{def}}{=} \text{the infinite set of all Erlang integers} \\
[\![\mathtt{float}()]\!] &\overset{\text{def}}{=} \text{the infinite set of all Erlang floats} \\
[\![\mathtt{number}()]\!] &\overset{\text{def}}{=} [\![\mathtt{integer}()]\!] \cup [\![\mathtt{float}()]\!] \\
[\![\mathtt{pid}()]\!] &\overset{\text{def}}{=} \text{the finite set of all Erlang PIDs} \\
[\![\mathtt{reference}()]\!] &\overset{\text{def}}{=} \text{the finite set of all Erlang references} \\
[\![\mathtt{port}()]\!] &\overset{\text{def}}{=} \text{the infinite set of all Erlang ports} \\
[\![\mathtt{list}()]\!] &\overset{\text{def}}{=} \text{the infinite set of all Erlang lists} \\
[\![\mathtt{tuple}()]\!] &\overset{\text{def}}{=} \text{the infinite set of all Erlang tuples}
\end{aligned}
$$

Literal types

$$
[\![\text{'}x\text{'}]\!] \overset{\text{def}}{=} \{\text{'}x\text{'}\}
$$

Compound types

$$
\begin{aligned}
[\![[\,]]\!] &\overset{\text{def}}{=} \{[\,]\} \\
[\![[\,T_h \mid T_t\,]]\!] &\overset{\text{def}}{=} \{t_h \in [\![T_h]\!], t_t \in [\![T_t]\!] \mid [\,h \mid t\,]\} \\
[\![\{\,\}]\!] &\overset{\text{def}}{=} \{\{\,\}\} \\
[\![\{T_1, T_2, \ldots, T_n\}]\!] &\overset{\text{def}}{=} \{t_1 \in [\![T_1]\!], t_2 \in [\![T_2]\!], \ldots, t_n \in [\![T_n]\!] \mid \{t_1, t_2, \ldots, t_n\}\}
\end{aligned}
$$

Type operators

$$
\begin{aligned}
[\![S \sqcup T]\!] &\overset{\text{def}}{=} [\![S]\!] \cup [\![T]\!] \\
[\![S \sqcap T]\!] &\overset{\text{def}}{=} [\![S]\!] \cap [\![T]\!] \\
[\![\neg T]\!] &\overset{\text{def}}{=} \mathcal{U} \setminus [\![T]\!]
\end{aligned}
$$

Figure 6.2: Denotational semantics for types

Theorem 6.2.1 (Associativity of $\sqcup$).

$$\forall S, T, U.\, [\![S \sqcup (T \sqcup U)]\!] = [\![(S \sqcup T) \sqcup U]\!]$$

Theorem 6.2.2 (Commutativity of $\sqcup$).

$$\forall S, T.\, [\![S \sqcup T]\!] = [\![T \sqcup S]\!]$$

Theorem 6.2.3 (Associativity of $\sqcap$).

$$\forall S, T, U.\, [\![S \sqcap (T \sqcap U)]\!] = [\![(S \sqcap T) \sqcap U]\!]$$

Theorem 6.2.4 (Commutativity of $\sqcap$).

$$\forall S, T.\, [\![S \sqcap T]\!] = [\![T \sqcap S]\!]$$

6.2.1 Sub-Typing

A relationship is starting to appear between types in the system: denotations of some types are subsets of the denotation of other types. For example, $[\![\mathtt{boolean}()]\!] \subseteq [\![\mathtt{atom}()]\!]$ (since the set {'true','false'} is a subset of the set of all atoms). This relationship is not coincidental: it reflects an observed behaviour of Erlang.

Table table 6.1 on the following page shows the relationships between primitive types by observing how Erlang's type test BIFs behave. In each case, assume an arbitrary assignment for the variable X. If the Erlang expression in the first column evaluates to 'true', then so does the expression in the second column. The third column shows the relationship between the types in the denotational semantics. Walking through the second row: if for some assignment of X the Erlang expression is_boolean(X) evaluates to 'true', then so does the expression is_atom(X). With respect to the denotational semantics, this relationship is represented as $[\![\mathtt{boolean}()]\!] \subseteq [\![\mathtt{atom}()]\!]$. In the cases when there is no *other* expression which always evaluates to 'true', the tautological expression true is used instead. In the denotational semantics, this is equivalent to the term() type as *all* Erlang values are member of term(), and the expression true returns 'true' for all assignments of X.

This relationship between types is called *sub-typing*:

Definition 6.2.1 (Sub-Typing ($\leqslant$)). *Type S is a sub-type of type T ($S \leqslant T$) if all members of S are also members of T. Using the denotational semantics, sub-typing is defined using the subset relation:*

$$S \leqslant T \stackrel{\text{def}}{=} [\![S]\!] \subseteq [\![T]\!]$$

This expression …	…implies this expression	Denotation
`is_atom(X)`	`'true'`	$[\![\mathit{atom}()]\!] \subseteq [\![\mathit{term}()]\!]$
`is_boolean(X)`	`is_atom(X)`	$[\![\mathit{boolean}()]\!] \subseteq [\![\mathit{atom}()]\!]$
`is_number(X)`	`'true'`	$[\![\mathit{number}()]\!] \subseteq [\![\mathit{term}()]\!]$
`is_float(X)`	`is_number(X)`	$[\![\mathit{float}()]\!] \subseteq [\![\mathit{number}()]\!]$
`is_integer(X)`	`is_number(X)`	$[\![\mathit{integer}()]\!] \subseteq [\![\mathit{number}()]\!]$
`is_list(X)`	`'true'`	$[\![\mathit{list}()]\!] \subseteq [\![\mathit{term}()]\!]$
`is_tuple(X)`	`'true'`	$[\![\mathit{tuple}()]\!] \subseteq [\![\mathit{term}()]\!]$
`is_pid(X)`	`'true'`	$[\![\mathit{pid}()]\!] \subseteq [\![\mathit{term}()]\!]$
`is_reference(X)`	`'true'`	$[\![\mathit{reference}()]\!] \subseteq [\![\mathit{term}()]\!]$
`is_port(X)`	`'true'`	$[\![\mathit{port}()]\!] \subseteq [\![\mathit{term}()]\!]$

Table 6.1: Observing Erlang's sub-typing axioms

Using this definition of sub-typing, each row of table 6.1 shows part of the sub-typing relationship, e.g. $\mathit{boolean}() \leqslant \mathit{atom}()$, $\mathit{float}() \leqslant \mathit{number}()$.

Furthermore, this definition of sub-typing is *semantic* rather than *syntactic*: the sub-typing relation is formulated using the denotational semantics of types (which is itself based on the operational semantics of CoErl), rather than being defined in terms of one or more syntactic rules.

This set-theoretic interpretation of types ultimately yields some "free" theorems: transitivity of $\leqslant$, identity elements for $\sqcup$ and $\sqcap$, and involution of $\neg$.

6.3 Sub-Typing Algorithm

The current definition of sub-typing uses a semantic model of types: the syntax of types is given meaning by a denotational semantics which associates a type with a set of values. This semantic model allows us to reason about types and the relationships between them using set theory, such as being able to represent the intersection of two types as a set, or represent the negation of a type as a complement of a set.

While this model of types (and sub-typing) is useful as part of an abstract formal model, it does not lend itself well to a succinct implementation in a general purpose programming language such as Erlang. Such an implementation would require significant bootstrapping: a denotational semantics for types, a general model of set theory, and symbolic handling of sets. An alternative approach is to manipulate the syntax of types directly using knowledge obtained from the semantic model. For example, the semantic model of types can be used to prove that

$\mathtt{term}()$ and $\neg\mathtt{term}()$ are the identity elements for intersection and union respectively:

$$\mathsf{T} \sqcap \mathtt{term}() = \mathsf{T}$$
$$\mathsf{T} \sqcup \neg\mathtt{term}() = \mathsf{T}$$

Essentially, we can use the denotational semantics for types to derive a set of rewrite rules for the *syntax* of types which preserve meaning.

Furthermore, we can note from set theory that our definition of sub-typing does not require the use of a subset judgement at all. Noting that subset is equivalent to checking for the empty set:

$$A \subseteq B \iff A \cap \overline{B} = \emptyset$$

we can redefine sub-typing using a similar check:

$$S \leqslant T \iff [\![S]\!] \subseteq [\![T]\!] \iff [\![S]\!] \cap \overline{[\![T]\!]} = \emptyset$$

An important part of this definition is that the equality check is taking place at the *semantic* level: we are checking whether the type on the left hand side is *semantically* equal to the empty set. This check must occur at the semantic level as our type syntax is not *canonical*: two types which are semantically equivalent can have a different syntax:

$$[\![\neg\neg T]\!] = [\![T]\!]$$
$$[\![S \sqcap T]\!] = [\![T \sqcap S]\!]$$

Pearce presents a method for performing this equality check at the syntactic level by performing a partial canonicalisation of type syntax. Observing that the sub-typing relation only holds when $[\![S \sqcap \neg T]\!]$ is equal to the empty set, the method focuses on canonicalising types which are semantically equal to the empty set, leaving all other types in some partly-normalised form.

In this section we present a variation of the original algorithm using CoErls type hierarchy and compound data types instead of the originals (Pearce 2013). The algorithm operates by first transforming all types into an enhanced disjunctive normal form wherein type operators are "lifted" out of compound types and all unions appear at the top level. Then, noting that it is relatively straightforward to write sound and complete intersection and sub-typing relations for a subset of the original type syntax, types are progressively rewritten until they are either $\neg\mathtt{term}()$, or some other type.

6.3.1 Positive Atoms

A central part of the sub-typing algorithm is the *positive type atom*: a type which does not contain any unions, intersections, or negations. These types are *positive* due to the absence

$$S^*, T^* = T^p \mid L \mid \mathit{nil} \mid [\, T^*_h \mid T^*_t \,] \mid \{\,\} \mid \{T^*_1, T^*_2, \dots, T^*_n\}$$

Figure 6.3: Positive type atom syntax

of any negations and they are *atomic* as they do not contain any unions or intersections. The presence of these type operators makes it difficult to define sub-typing or intersection using a purely inductive or syntactically recursive definition, as corner cases such as double negations, nesting of operators inside compound types, and the commutativity of union and intersection all require special consideration. By omitting these operators, we can write straightforward recursive definitions of type intersection and sub-typing which can be easily proven correct with respect to the denotational semantics.

The syntax of positive type atoms is defined as T^* in figure 6.3. This is a restriction of the original syntax of types T from figure 6.1 on page 95. In this restricted syntax types must be primitive (T^p, e.g. $\mathit{atom}()$, $\mathit{integer}()$), singletons (L, e.g. 2.0, `'foo'`), or a compound type constructor consisting of these types (e.g. $[\, \mathit{integer}() \mid [\,]\,]$).

Intersection

The intersection of two types is the type of values which inhabits both types. For example, the intersection of $\mathit{atom}()$ and $\mathit{boolean}()$ is $\mathit{boolean}()$, and the intersection of $\mathit{atom}()$ and $\mathit{integer}()$ is $\neg\mathit{term}()$ (as there are no values which inhabit both types).

To intersect two positive type atoms, we use the infix $\sqcap^*$ operator from figure 6.4 on the next page. This should not be confused with the syntactic $\sqcap$ seen in types.

First, we consider whether two types are syntactically equal. If they are, then we return one of the two types (6.1) (as $S = T \Rightarrow [\![S]\!] = [\![T]\!]$). Next, we encode the axioms of sub-typing: $\mathit{boolean}()$ is a sub-type of $\mathit{atom}()$ therefore the intersection is $\mathit{boolean}()$ (6.3), $\mathit{integer}()$ is a sub-type of $\mathit{number}()$ so the intersection is $\mathit{integer}()$ (6.4), and similarly for $\mathit{float}()$ (6.5), $[\,]$ (6.6), and $\{\,\}$ (6.9). Also, any cons cell is a sub-type of $\mathit{list}()$, so the intersection of the two is the cons cell (6.7), and similarly for tuples (6.10). When both types being intersected are cons cells, we intersect the heads and tails of both cells separately. Assuming that both results are not $\neg\mathit{term}()$, then the intersection of the two cons cells is a cons cell consisting of the intersections of the heads and tails of the original two cells (6.8). A similar rule applies to two non-empty tuples: assuming that they have the same number of elements and that each element of the first intersects with the corresponding element in the second, then the result is a tuple containing all of the intersected elements (6.11). In all other cases, the intersection of the two types is the empty

$$
\begin{align}
T \sqcap^* T &\Longrightarrow T \tag{6.1}\\
S \sqcap^* \mathtt{term}() &\Longrightarrow S \tag{6.2}\\
\mathtt{atom}() \sqcap^* \mathtt{boolean}() &\Longrightarrow \mathtt{boolean}() \tag{6.3}\\
\mathtt{number}() \sqcap^* \mathtt{integer}() &\Longrightarrow \mathtt{integer}() \tag{6.4}\\
\mathtt{number}() \sqcap^* \mathtt{float}() &\Longrightarrow \mathtt{float}() \tag{6.5}\\
\mathtt{list}() \sqcap^* [\,] &\Longrightarrow [\,] \tag{6.6}\\
\mathtt{list}() \sqcap^* [\, T_h \mid T_t \,] &\Longrightarrow [\, T_h \mid T_t \,] \tag{6.7}\\
[\, S_h \mid S_t \,] \sqcap^* [\, T_h \mid T_t \,] &\Longrightarrow [\, S_h \sqcap^* T_h \mid S_t \sqcap^* T_t \,] \tag{6.8}\\
\mathtt{tuple}() \sqcap^* \{\,\} &\Longrightarrow \{\,\} \tag{6.9}\\
\mathtt{tuple}() \sqcap^* \{T_1, T_2, \ldots, T_n\} &\Longrightarrow \{T_1, T_2, \ldots, T_n\} \tag{6.10}\\
\{S_1, S_2, \ldots, S_n\} \sqcap^* \{T_1, T_2, \ldots, T_n\} &\Longrightarrow \{S_1 \sqcap^* T_1, S_2 \sqcap^* T_2, \ldots, S_n \sqcap^* T_n\} \tag{6.11}
\end{align}
$$

All cases are symmetric.

Figure 6.4: Intersection for atomic types ($\sqcap^*$)

type $\neg\mathtt{term}()$, but we must consider that intersection is a symmetric relation: $A \sqcap B = B \sqcap A$ for all A and B, hence some rules are symmetric.

Lemma 6.3.1. *For all S^* and T^*:*

$$[\![S^* \sqcap^* T^*]\!] = [\![S^*]\!] \cap [\![T^*]\!]$$

Proof. Straightforward by inspection of figure 6.2 and figure 6.4. □

Sub-Typing

Sub-typing can also be defined in a way that is both sound and complete with respect to the denotational semantics. The sub-typing relation will be defined inductively, making it straightforward to encode the transitive property of the relation.

The definition of sub-typing for positive type atoms is given in figure 6.5 on the following page, and is written using the infix notation $\leqslant^*$ which should not be confused with the semantic sub-typing operator which lacks a star ($\leqslant$).

The first two rules represent the reflexivity and transitivity of sub-typing (Refl and Trans). Next, all types are sub-types of the top type $\mathtt{term}()$: as $\mathtt{term}()$ represents the universe of all values, all other types must be sub-types (Term). The rules Boolean, Float, Nil, and TupleNil are axioms of sub-typing, similar to those we saw for atomic type intersection.

A cons cell is a sub-type of another cons cell if the head of the first is a sub-type of the head of the second, and also if the tail of the first is a sub-type of the tail of the second, i.e. the relation

$$\frac{S^* = T^*}{S^* \leqslant^* T^*}\ \textsc{Refl} \qquad \frac{S^* \leqslant^* U^* \qquad U^* \leqslant^* T^*}{S^* \leqslant^* T^*}\ \textsc{Trans} \qquad \frac{}{S^* \leqslant^* \mathtt{term}()}\ \textsc{Term}$$

$$\frac{}{\mathtt{boolean}() \leqslant^* \mathtt{atom}()}\ \textsc{Boolean} \qquad \frac{}{\mathtt{float}() \leqslant^* \mathtt{number}()}\ \textsc{Float}$$

$$\frac{}{\mathtt{integer}() \leqslant^* \mathtt{number}()}\ \textsc{Integer} \qquad \frac{}{[\,] \leqslant^* \mathtt{list}()}\ \textsc{Nil} \qquad \frac{}{[\,S_h^* \mid S_t^*\,] \leqslant \mathtt{list}()}\ \textsc{ConsLeft}$$

$$\frac{S_h^* \leqslant^* T_h^* \qquad S_t^* \leqslant^* T_t^*}{[\,S_h^* \mid S_t^*\,] \leqslant^*}\ \textsc{Cons} \qquad \frac{}{\{\,\} \leqslant^* \mathtt{tuple}()}\ \textsc{TupleNil} \qquad \frac{n \neq 0}{\{S_1^*, S_2^*, \ldots, S_n^*\}}\ \textsc{TupleLeft}$$

$$\frac{S_1^* \leqslant^* T_1^* \qquad S_2^* \leqslant^* T_2^* \qquad S_n^* \leqslant^* T_n^*}{\{S_1^*, S_2^*, \ldots, S_n^*\} \leqslant^* \{T_1^*, T_2^*, \ldots, T_n^*\}}\ \textsc{TupleN}$$

Figure 6.5: Sub-typing for atomic types ($\leqslant^*$)

distributes over the elements of the cell (Cons). Similarly, a tuple of size n is a sub-type of another tuple with size n if and only if each element of the first tuple is a sub-type of the corresponding element of the second tuple (TupleN).

Lemma 6.3.2. *For all S^* and T^*:*

$$S^* \leqslant^* T^* \iff [\![S^*]\!] \subseteq [\![T^*]\!]$$

Proof. Straightforward by inspection of figure 6.5 and figure 6.2. □

6.3.2 Disjunctive Normal Form

We now have sound and complete definitions of intersection ($\sqcap^*$) and sub-typing ($\leqslant^*$). Unfortunately, these definitions are only sound and complete *up to a subset of the original type syntax*: only positive type atoms are supported, i.e. there can not be any unions, intersections, or negations.

Regardless of this limitation, we can use these definitions to simplify *some parts* of types, regardless of their overall syntactic structure. For example, if the syntax of a term contains an intersection of two positive type atoms $S^* \sqcap T^*$, we can replace it with the result of $S^* \sqcap^* T^*$ without changing the semantics of the type (as per lemma 6.3.1).

As we can easily simplify types containing intersections of positive type atoms, it stands to reason that we could greatly simplify types containing a large number of these intersections. To maximise the occurrence of them we will rewrite types into a normal form similar to a disjunctive normal form. Specifically, we will rewrite all types of syntax T (figure 6.1) into syntax T^{DNF} (figure 6.6 on the next page): unions at the top level, followed by intersections of (possible

$$T^{\sqcap} ::= T^* \mid \neg T^* \mid T^{\sqcap} \sqcap T^{\sqcap}$$
$$T^{\mathrm{DNF}} ::= T^{\sqcap} \mid T^{\sqcap} \sqcup T^{\sqcap}$$

Figure 6.6: DNF Grammar for types

$$(S_1^* \sqcup S_2^*) \sqcap T^* \Longrightarrow (S_1^* \sqcap T^*) \sqcup (S_2^* \sqcap T^*) \tag{6.12}$$
$$\neg\neg T \Longrightarrow T \tag{6.13}$$
$$[\,\neg T_h^* \mid T_t^*\,] \Longrightarrow [\,\mathtt{term}() \mid T_t^*\,] \sqcap \neg [\,T_h^* \mid T_t^*\,] \tag{6.14}$$
$$[\,T_h^* \mid \neg T_t^*\,] \Longrightarrow [\,T_h^* \mid \mathtt{term}()\,] \sqcap \neg [\,T_h^* \mid T_t^*\,] \tag{6.15}$$
$$[\,S_h^* \text{ op } T_h^* \mid T_t^*\,] \Longrightarrow [\,T_h^* \mid S_t^*\,] \text{ op } [\,T_h^* \mid T_t^*\,] \quad \text{where op is } \sqcap \text{ or } \sqcup \tag{6.16}$$
$$[\,S_h^* \mid S_t^* \text{ op } T_t^*\,] \Longrightarrow [\,S_h^* \mid T_t^*\,] \text{ op } [\,T_h^* \mid T_t^*\,] \quad \text{where op is } \sqcap \text{ or } \sqcup \tag{6.17}$$
$$\{T_1^*, \neg T_m^*, \ldots, T_n^*\} \Longrightarrow \{T_1^*, \mathtt{term}(), \ldots, T_m^*\} \sqcap \neg \{T_1^*, T_m^*, \ldots, T_n^*\} \tag{6.18}$$
$$\{T_1^*, S_m^* \text{ op } T_m^*, \ldots, T_n^*\} \Longrightarrow \{T_1^*, S_m^*, \ldots, T_n^*\} \text{ op } \{T_1^*, T_m^*, \ldots, T_n^*\} \tag{6.19}$$
$$\tag{6.20}$$

Figure 6.7: Normalisation rules for T

negations of) positive type atoms. For example, the type $(\mathtt{atom}() \sqcap \mathtt{boolean}()) \sqcup (\mathtt{integer}() \sqcap \neg\mathtt{pid}())$ is in normal form as all unions appear at the top level and all intersections are (perhaps negations of) type atoms, but $(\mathtt{atom}() \sqcup \mathtt{integer}()) \sqcap \mathtt{reference}()$ is not as there is a union inside an intersection. In addition, note that T^{DNF} does not permit type operators inside compound types as per the syntax of positive type atoms: there cannot be any unions, intersections, or negations inside cons cells or tuples.

The rewrite rules in figure 6.7 are used to transform a type of syntax T into the syntax T^{DNF} while preserving the semantics of the original type.

The first rule (6.12) relies on the fact that intersection distributes over union:

$$[\![(S \sqcup T) \sqcap U]\!] \iff ([\![S]\!] \cup [\![T]\!]) \cap [\![U]\!] \iff ([\![S]\!] \cap [\![U]\!]) \cup ([\![T]\!] \cap [\![U]\!]) \iff [\![(S \sqcap U) \sqcup (T \sqcap U)]\!]$$

Rules (6.14) and (6.15) handle negations in the heads and tails of cons cells. For heads, the rewrite rule states that any cons cell $[\,\neg S \mid T\,]$ is equivalent to an intersection of two cells: where the head can be any value and the tail must have type T ($[\,\mathtt{term}() \mid T\,]$), and *not* the cons cells where the head has type S and the tail has type T. A similar rule exists for tuples which operates on any element, including the first and last (6.18). Furthermore, operators ($\sqcap$ and $\sqcup$) are lifted out of cons cells and tuples using rules (6.17), (6.16), and (6.19).

$$
\begin{aligned}
\neg\mathtt{term}() \sqcap \ldots &\Longrightarrow \neg\mathtt{term}() && && (6.21)\\
S^* \sqcap T^* \sqcap \ldots &\Longrightarrow (S^* \sqcap^* T^*) \sqcap \ldots && && (6.22)\\
S^* \sqcap \neg T^* \sqcap \ldots &\Longrightarrow \neg\mathtt{term}() && \text{if } S^* \leqslant^* T^* && (6.23)\\
S^* \sqcap \neg T^* \sqcap \ldots &\Longrightarrow S^* \sqcap \ldots && \text{if } S^* \sqcap^* T^* = \neg\mathtt{term}() && (6.24)\\
S^* \sqcap \neg T^* \sqcap \ldots &\Longrightarrow S^* \sqcap \neg(S^* \sqcap^* T^*) \sqcap \ldots && \text{if } S^* \not\geqslant^* T^* && (6.25)\\
\neg S^* \sqcap \neg T^* \sqcap \ldots &\Longrightarrow \neg S \sqcap \ldots && \text{if } S^* \geqslant^* T^* && (6.26)
\end{aligned}
$$

All rules are symmetric.

Figure 6.8: Canonicalisation rules for positive type atoms in disjunctive normal form

Finally, we define the function DNF which converts a type with syntax T to one with syntax T^*:

Definition 6.3.1. *The function* $\mathsf{DNF}(T)$ *is the exhaustive application of rewrite rules from figure 6.7 to the type* T*.*

The rewrite rules form a function via outermost application, starting from the left. Although the rules are confluent, applying them in arbitrary order could lead to different outputs for different inputs because of the commutativity and associativity of $\sqcap$ and $\sqcup$.

Lemma 6.3.3. *For all types* T*:*

$$[\![T]\!] = [\![\mathsf{DNF}(T)]\!]$$

Proof. Sketch:

- Prove that each rewrite rule preserves meaning (any rewrite is semantically equivalent).
- Prove that repeated rewrites also prove meaning (reflexive and transitive).
- Induction on T.

□

Lemma 6.3.4. *For all types* T*,* $\mathsf{DNF}(T)$ *returns a type of syntax* T^**.*

6.3.3 Canonicalisation

As mentioned earlier, it is relatively easy to calculate the intersection of two positive types compared to doing the same with two types of arbitrary syntax (which could contain union, intersection, and negation in arbitrary locations). By transforming a type into disjunctive normal form using the DNF function from the previous section, we obtain a semantically equivalent type whose syntax is of the form $\bigsqcap \bigsqcup \mathsf{T}^{*-}$, where T^{*-} can be positive type atoms and/or negations thereof.

At this point we can use the intersection function $\sqcap^*$ and sub-typing operator $\leqslant^*$ to canonicalise types in disjunctive normal form. Exhaustive application of the rewrite rules in figure 6.8 on the previous page will result in a type which is either a union $\neg\mathtt{term}()$ types, or some other type. As we will see, this is enough to implement sub-typing.

Rule (6.21) in figure 6.8 handles an intersection with the empty $\neg\mathtt{term}()$ type: any intersection with the empty set is itself empty, hence we replace the entire intersection with $\neg\mathtt{term}()$. The rule (6.22) handles an intersection of two positive type atoms as part of a larger intersection: the result is the type returned by the application of $\sqcap^*$ from figure 6.4 on page 103.

An intersection where exactly one of the two operands is negative $(\mathsf{S} \sqcap \neg\mathsf{T})$ requires careful consideration: if S is a sub-type of T, then there are no elements in S which are not also members of T (6.23). If S is not a sub-type of T, then we must consider how the two types intersect: if there is no overlap between the two types, then we can discard $\neg\mathsf{T}$ entirely, and we do not rewrite it otherwise. Finally, if we have an intersection of two negative type atoms, we follow the same scheme as rule (6.22), but with the arguments to $\leqslant^*$ flipped: if T is a sub-type of S, then $\neg\mathsf{S}$ is superfluous, as $\neg\mathsf{T}$ already excludes all members of $\neg\mathsf{S}$ (6.26).

Similar to the DNF function, we exhaustively apply these rules to a type until it cannot be rewritten any further, starting from the top and working from the left. Again, these rules are confluent with respect to the denotational semantics of types, but we apply an order to the rewrites so that the *syntax* produced is canonical. This rewrites every conjunction $\mathsf{S} \sqcap \mathsf{T} \sqcap \mathsf{U} \sqcap \ldots$ to one of two forms:

1. the type $\neg\mathtt{term}()$; or
2. some intersection of (potentially negatated) positive type atoms

By exhaustively rewriting a type that is already in DNF form, we will therefore obtain a type which is either a union of one or more $\neg\mathtt{term}()$ types, or a union of intersections.

Definition 6.3.2. *The function* $\mathsf{CAN}(\mathsf{T})$ *is the exhaustive application of rewrite rules from figure 6.8 to the type* T.

Lemma 6.3.5. *For all types* T*:*

$$[\![\mathsf{T}]\!] = [\![\mathsf{CAN}(\mathsf{T})]\!]$$

Proof. Sketch:

- Prove that canonicalisation rules preserve meaning w.r.t. semantics.
- Induction on T

$\square$

Lemma 6.3.6. *For all types* T *of the form* $\bigsqcup\bigsqcap \mathsf{T}^{*-}$, $\mathsf{CAN(T)}$ *is either of the form* $\bigsqcup\neg\mathtt{term}()$ *or* $\bigsqcup\bigsqcap \mathsf{T}^{*-}$.

6.3.4 Enhanced Disjunctive Normal Form

The final step in normalising types is to convert them into an *enhanced* disjunctive normal form: types are first converted into DNF, then canonicalised:

Definition 6.3.3.

$$\mathsf{DNF}^{+}(\mathsf{T}) = \mathsf{CAN}(\mathsf{DNF}(\mathsf{T}))$$

Theorem 6.3.1. *For all types* T*:*

$$[\![\mathsf{T}]\!] = [\![\mathsf{DNF}^{+}(\mathsf{T})]\!]$$

Proof. Straightforward by lemma 6.3.4 and lemma 6.3.5. $\square$

6.3.5 Sub-Typing

Finally, we bring all of these definitions to define sub-typing. First, we recall that sub-typing is equivalent to a problem of checking whether an intersection is equal to the empty set:

$$\mathsf{S} \leqslant \mathsf{T} \iff [\![\mathsf{S}]\!] \subseteq [\![\mathsf{T}]\!] \iff [\![\mathsf{S}]\!] \cap \overline{[\![\mathsf{T}]\!]} = \emptyset \iff [\![\mathsf{S} \sqcap \neg\mathsf{T}]\!] = \emptyset$$

Therefore, checking whether $\mathsf{S} \leqslant \mathsf{T}$ is equivalent to checking whether $\mathsf{S} \sqcap \neg\mathsf{T}$ denotes the empty set. Previously this check was performed at the *semantic* level, but we can now perform the same check at the *syntactic* level using DNF^{+}, which normalises a type to a union of $\neg\mathtt{term}()$ types if it is semantically equivalent to the empty set:

Definition 6.3.4. *For all types* S *and* T, S *is a sub-type of* T *if* $\mathsf{DNF}^{+}(\mathsf{S} \sqcap \neg\mathsf{T})$ *returns a union of the type* $\neg\mathtt{term}()$

$$\mathsf{S} \leqslant \mathsf{T} \iff \mathsf{DNF}^{*}(\mathsf{S} \sqcap \neg\mathsf{T}) = \bigsqcup\neg\mathtt{term}()$$

As an example of this algorithm, we will consider whether $\mathsf{term}()$ is a sub-type of $\mathsf{integer}() \sqcup \neg\mathsf{integer}()$. In the denotational semantics, the sub-typing relation holds:

$$\begin{aligned}\mathsf{term}() \leqslant \mathsf{integer} \sqcup \neg\mathsf{integer}() \iff & [\![\mathsf{term}()]\!] \subseteq [\![\mathsf{integer}() \sqcup \neg\mathsf{integer}]\!] \\ &= \mathcal{U} \subseteq [\![\mathsf{integer}()]\!] \cup \overline{[\![\mathsf{integer}()]\!]} \\ &= \mathcal{U} \subseteq \mathcal{U}\end{aligned}$$

It also holds using our canonicalisation algorithm:

$$\begin{aligned}&\mathsf{DNF}^{+}(\mathsf{term}() \sqcap \neg(\mathsf{integer}() \sqcup \neg\mathsf{integer}())) = \bigsqcup \neg\mathsf{term}() \\ &= \mathsf{CAN}(\mathsf{DNF}(\mathsf{term}() \sqcap \neg(\mathsf{integer}() \sqcap \neg\mathsf{integer}()))) = \bigsqcup \neg\mathsf{term}() \\ &= \mathsf{CAN}(\mathsf{any}() \sqcap \neg\mathsf{integer}() \sqcap \mathsf{integer}()) = \bigsqcup \neg\mathsf{term}() \\ &= \neg\mathsf{term} = \bigsqcup \neg\mathsf{term}()\end{aligned}$$

Although this algorithm provides a sound and complete method for determining the sub-typing relation, it can be slow due to its exponential blowup. To this end, chapter 7 introduces an alternative (and less complex) algorithm based on *Binary Decision Diagrams* (BDDs) which operates using a similar principle: producing a canonical data structure for a type, and checking whether it is structurally identical to the canonical form of $\neg\mathsf{term}()$ in the system.

6.4 Type Inference

To reason about the types of messages a process can receive, static analysis will be performed on the patterns, guards, and clauses of receive expressions present in the process's code. Patterns and guards inherently contain type information about the types of values they accept, a fact which will be exploited to perform fully automatic type inference on Erlang code.

For example, the following Erlang function calculates the length of a proper list by pattern matching on its sole argument:

```erlang
length([]) -> 0;
length([_H|T]) -> 1 + length(T).
```

By intuition we might assign the type $\mathsf{list}()$ to the argument of the length function: the pattern [ ] only matches the empty list constructor, and the pattern [_H | T] in the second clause only matches cons cells. As the function can accept *either* of these list constructors, we might say that the argument *must* have type $[\,] \sqcup [\,\mathsf{term}() \mid \mathsf{term}()\,]$ (or $\mathsf{list}()$ for brevity). Type information is also contained in guard expressions, where type assertions can be derived from equality checks

$$\mathcal{P}\,[\![p]\!]_\Gamma \overset{\text{def}}{=} \begin{cases} T & \text{if } p = v \text{ and } \Gamma \vdash v : T \\ \mathcal{P}\,[\![p']\!]_\Gamma \sqcap T & \text{if } p = (p' = v) \text{ and } \Gamma \vdash v : T \\ [\,] & \text{if } p = [\,] \\ [\,\mathcal{P}\,[\![p_h]\!]_\Gamma \mid \mathcal{P}\,[\![p_t]\!]_\Gamma\,] & \text{if } p = [\,p_h \mid p_t\,] \\ \{\,\} & \text{if } p = \{\,\} \\ \{\mathcal{P}\,[\![p_1]\!]_\Gamma, \mathcal{P}\,[\![p_2]\!]_\Gamma, \ldots, \mathcal{P}\,[\![p_n]\!]_\Gamma\} & \text{if } p = \{p_1, p_2, \ldots, p_n\} \end{cases}$$

Figure 6.9: Type inference for CoErl patterns

and the use of type test BIFs. This function, for example, requires that its argument is a 2-element tuple where the first element is the atom `'double'` and the second element is an integer:

```
do_op({'double',N}) when is_integer(N) -> N + N.
```

The type of this function's argument is written $\{\text{‘'double'’}, \mathtt{integer}()\}$.

Finally, the order of clauses must be considered, as the patterns and guards of *prior* clauses affect the types of values accepted by a clause:

```
foo(N) when is_integer(N) -> "it's an integer!";
foo(N) -> "it's not an integer".
```

In this example the first clause accepts only integers ($\mathtt{integer}()$), while the second clause accepts values which are *not* integers ($\neg\mathtt{integer}()$).

Together, these three principles form the basis of type inference for CoErl patterns, guards, and clauses.

6.4.1 Patterns

Type inference on patterns is performed by recursion over the structure of the pattern, with the aim of determining the types of values that will match the pattern in question. As defined in figure 4.1, a pattern p can be a variable v, an alias $p = v$, a list ($[\,]$ or $[\,p_h \mid p_t\,]$), or a tuple ($\{\,\}$ or $\{p_1, p_2, \ldots, p_n\}$).

The inference is performed in a typing environment Γ which associates every variable with a type, such that $\Gamma \vdash v : T$ means that variable v has type T in environment Γ. The function $\mathcal{P}\,[\![p]\!]_\Gamma$ in figure 6.9 performs type inference on a pattern p in typing environment Γ, where the output of the function is a type.

Starting with the first case, a variable v is inferred to have type T, where T is the type provided by the typing environment. Variables also occur in *aliases* $(p' = v)$, where a pattern p is associated with a variable v. In this case, the inferred type is an intersection of the inferred type of p' and the type of v from the typing environment: the type of $p' = v$ is the inferred type of p' *and* the type of v from the environment.

Moving on to compound patterns, the inference proceeds by recursing over the structure of the type. For empty lists ($[\,]$), the inferred type is the empty list type which shares the same syntax. For cons cells of the form $[\,p_h \mid p_t\,]$, the types of the head and tail are inferred recursively. Similar rules apply for tuples: the empty tuple $\{\,\}$ has an inferred type of $\{\,\}$, and non-empty tuples have their types inferred recursively.

6.4.2 Guards

Type inference for guards is more complicated than type inference for patterns: the syntax of guards allows more complex type constraints to be expressed. A guard expression is typically used to impose additional constraints on the type or value of variables which occur in an accompanying pattern. For example, a guard expression in Erlang can be used to assert that a variable is a member of at least one of two types:

```
f({A,B}) when is_integer(A) or is_atom(A) -> do_stuff.
```

The syntax of guards in CoErl is more restrictive, instead only permitting four constructs: the atoms `'true'` and `'false'`, an assertion that a variable v has type T (v `is` T), and an if-then-else expression which can be used to build logical conjunction, disjunction, and negation (**if** $g_?$ **then** g_t **else** g_f). The branching evaluation of the if-then-else expression is the source of complexity for guard type inference due to the presence of multiple control flow paths. Consider the following guard expression:

$$\textbf{if } v \text{ is } T \textbf{ then } w \text{ is } U \textbf{ else } w \text{ is } V$$

The evaluation of the guard succeeds (i.e. it evaluates to true) if and only if:

1. v has type T *and* w has type U; *or*

2. v does not have type T *and* w has type V.

Considering that a typing environment returned from an inference algorithm should state the type constraints on variables present in the guard, a reasonable approximation might be:

$$v \mapsto T,\ w \mapsto U \sqcup V$$

Unfortunately, this is a bad approximation: it simultaneously contains type constraints not present in the guard and also does not correctly constrain the types of all variables. The typing environment over-approximates the type constraints on v, which is asserted to have type T in the environment. The guard evaluation can succeed, however, even if v does not have type T: when w has type V. On the other hand, the environment under-approximates the type constraints on w, where the environment states that w has either type U or V. Assuming that v has type T, though, guard evaluation will not succeed unless w has type U.

A single typing environment is not enough to specify the type constraints of *multiple* control flow branches. Each branch can have entirely different type constraints: the types of different variables may be asserted in different branches and some variables may not have type assertions at all. Instead of a single environment for all control flow branches, an environment will be created for *each* branch. As the original guard contains two branches of execution (the true and false branches of the if-then-else), we will create two typing environments:

1. $[v \mapsto \mathsf{T},\ w \mapsto \mathsf{U}]$
2. $[v \mapsto \neg\mathsf{T},\ w \mapsto \mathsf{V}]$

The first environment asserts that v has type T and w has type U, while the second asserts that v *does not* have type T and w has type V. This corresponds more closely with our intuition about how the guard evaluates: either the test of the if-then-else succeeds (and the true branch is evaluated) or it does not (and the false branch is evaluated). In the first case, we know that v is T is true, and in the second case we know it is *not* true, so we can assert that $v : \neg\mathsf{T}$. Overall, the guard type inference algorithm needs to return multiple typing environments: one for each control flow path which results in the successful evaluation of the guard.

One additional complication is that the test of an if-then-else guard may also be an if-then-else. Let's consider the guard from earlier, but wrapped in another if-then-else:

$$\textbf{if } (\textbf{if } v \text{ is } \mathsf{T} \textbf{ then } w \text{ is } \mathsf{U} \textbf{ else } w \text{ is } \mathsf{V}) \textbf{ then } x \text{ is } \mathsf{S} \textbf{ else } x \text{ is } \mathsf{S}'$$

To avoid confusion, the outermost if-then-else (**if** ... **then** x is S **else** x is S') will be called the *outer*, and the innermost (**if** ... **then** w is U **else** w is V) will be called the *inner*. This guard expression will evaluate to true if and only if:

1. the inner guard evaluates to true *and* x is S returns true; *or*
2. the inner guard evaluates to false *and* x is S' returns true.

Creating type constraints for the first case is straightforward: the type constraints from the inner if-then-else ($[v \mapsto \mathsf{T},\ w \mapsto \mathsf{U}]$ and $[v \mapsto \neg\mathsf{T},\ w \mapsto \mathsf{V}]$) can be concatenated with the type constraints from x is S ($[x \mapsto \mathsf{S}]$). This concatenation is done with the $\wedge_{[\Gamma]}$ operator from figure 6.10. The operator creates a conjunction of two lists of typing environment based on the assumption that each list of environments represents a disjunction. For example, this list of environments:

$$[\,[v \mapsto T,\ w \mapsto U],\ [v \mapsto \neg T,\ w \mapsto V]\,]$$

represents a disjunction of two possibilities: either the typing constraints from the first environment hold, or those from the second hold. As such, the $\wedge_{[\Gamma]}$ operator from figure 6.10 performs a task similar to converting a boolean formula into a disjunctive normal form:

$$(A \vee B) \wedge (C \vee D) = (A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (C \wedge D)$$

As for the conjunction of two individual environments ($\Gamma \wedge_\Gamma \Gamma'$), the resulting environment depends on whether a given variable appears in one or both environments:

$$\begin{aligned} [x \mapsto T] \wedge_\Gamma [\,] &= [x \mapsto T] \\ [\,] \wedge_\Gamma [x \mapsto T] &= [x \mapsto T] \\ [x \mapsto S] \wedge_\Gamma [x \mapsto T] &= [x \mapsto S \sqcap T] \\ [x \mapsto S] \wedge_\Gamma [y \mapsto T] &= [x \mapsto S,\ y \mapsto T] \end{aligned}$$

If a variable is present in only one of the environments, it is present in the resulting environment unchanged. If both environments contain the variable, the intersection of that variable's type are used.

Going back to the example on the preceding page: we know the inner if-then-else evaluates to true when $[v \mapsto T,\ w \mapsto U]$ or $[v \mapsto \neg T,\ w \mapsto V]$, while the true branch of the outer if-then-else evaluates to true if $[x \mapsto S]$. The conjunction of these two lists of environments gives the type constraints for when the true branch evaluates to true:

$$\begin{aligned} [\,[v \mapsto T, w \mapsto U],\ [v \mapsto \neg T, w \mapsto V]\,] \wedge_{[\Gamma]} [\,[x \mapsto S]\,] = [\,&[v \mapsto T, w \mapsto U, x \mapsto S], \\ &[v \mapsto \neg T, w \mapsto V, x \mapsto S]\,] \end{aligned}$$

Considering the type constraints for the false branch of the outer if-then-else is less straightforward: the only way to ever evaluate the false branch is to have followed a control flow path of the inner if-then-else which evaluated to false. Therefore, we need to know which type constraints lead to the innermost if-then-else evaluating to false:

$$\textbf{if } v \text{ is } T \textbf{ then } w \text{ is } U \textbf{ else } w \text{ is } V$$

There are two control flow paths where this guard evaluates to false:

Conjunction of two environments

$$\Gamma \wedge_\Gamma \Gamma' \overset{\text{def}}{=} \lambda x. \begin{cases} \text{Some } T \sqcap T' & \text{if } \Gamma(x) = \text{Some } T \text{ and } \Gamma'(x) = \text{Some } T' \\ \text{Some } T & \text{if } \Gamma(x) = \text{Some } T \text{ and } \Gamma'(x) = \text{None} \\ \text{Some } T' & \text{if } \Gamma(x) = \text{None and } \Gamma'(x) = \text{Some } T' \\ \text{None} & \text{otherwise} \end{cases}$$

Conjunction of two lists of environments

$$\begin{aligned}[\Gamma_1, \Gamma_2, \ldots, \Gamma_m] \wedge_{[\Gamma]} [\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_n] \overset{\text{def}}{=} [\, & \Gamma_1 \wedge_\Gamma \Gamma'_1,\ \Gamma_1 \wedge_\Gamma \Gamma'_2,\ \ldots,\ \Gamma_1 \wedge_\Gamma \Gamma'_n, \\ & \Gamma_2 \wedge_\Gamma \Gamma'_1,\ \Gamma_2 \wedge_\Gamma \Gamma'_2,\ \ldots,\ \Gamma_2 \wedge_\Gamma \Gamma'_n, \\ & \vdots \\ & \Gamma_m \wedge_\Gamma \Gamma'_1,\ \Gamma_m \wedge_\Gamma \Gamma'_2,\ \ldots,\ \Gamma_m \wedge_\Gamma \Gamma'_n \,]\end{aligned}$$

Figure 6.10: Operators for conjunction of typing environments

$$\mathcal{G}\,[\![g]\!] \overset{\text{def}}{=} \begin{cases} ([\lambda x.\text{Some } \mathtt{term}()], []) & \text{if } g = \mathtt{'true'} \\ ([], [\lambda x.\text{Some } \mathtt{term}()]) & \text{if } g = \mathtt{'false'} \\ ((g_?^+ \wedge_{[\Gamma]} g_t^+) +\!\!+ (g_?^- \wedge_{[\Gamma]} g_f^+), & \text{if } g = \textbf{if } g_? \textbf{ then } g_t \textbf{ else } g_f \\ \quad (g_?^+ \wedge_{[\Gamma]} g_t^-) +\!\!+ (g_?^- \wedge_{[\Gamma]} g_f^-)) & \text{where} \\ & \quad \mathcal{G}\,[\![g_?]\!] = (g_?^+, g_?^-) \\ & \quad \mathcal{G}\,[\![g_t]\!] = (g_t^+, g_t^-) \\ & \quad \mathcal{G}\,[\![g_f]\!] = (g_f^+, g_f^-) \\ ([[v \mapsto T]], [[v \mapsto \neg T]]) & \text{if } g = v \ \mathtt{is}\ T \end{cases}$$

Figure 6.11: Type inference for CoErl guards

1. when v is T evaluates to true *and* w is U evaluates to false; *or*

2. when v is T evaluates to false *and* $w$is V evaluates to false

In other words, these are the typing environments in which the guard evaluates to false:

1. $[v \mapsto T, w \mapsto \neg U]$

2. $[\mapsto \neg T, w \mapsto \neg V]$

Note that these typing environments are *not* simply the negations of the environments from the true branches.

This insight into the control flow of guard expressions has been used to write the $\mathcal{G}\,[\![\cdot]\!]$ function in figure 6.11, which performs type inference for guard expressions. The function simultaneously

$$\mathcal{C}\,[\![\langle p\rangle \text{ when } g \to e]\!] \overset{\text{def}}{=} \textbf{let } \Gamma_1, \Gamma_2, \ldots, \Gamma_n = \mathtt{fst}(\mathcal{G}\,[\![g]\!]) \textbf{ in}$$
$$\mathcal{P}\,[\![p]\!]_{\Gamma_1} \sqcup \mathcal{P}\,[\![p]\!]_{\Gamma_2} \sqcup \ldots \sqcup \mathcal{P}\,[\![p]\!]_{\Gamma_n}$$

Figure 6.12: Type inference for CoErl clauses

produces two lists of typing environments: the environments in which the guard evaluates to true, and the environments in which the guard evaluates to false.

The guard `'true'` returns true regardless of the types of any variables, so the environments in which it returns true are those where all variables have the $\mathtt{term}()$ type; it does not matter what the type of a variable is. Furthermore, there are no environments in which the guard evaluates to false. Conversely, the guard `'false'` returns false in all cases, so it returns true for no environments and returns false regardless of the types of variables.

When an if-then-else is encountered, the function is recursive. First, the true and false environments for the test $g_?$ are inferred, followed by g_t and g_f. As discussed, the circumstances in which this if-then-else returns true are when either both the test $g_?$ and true branch g_t evaluate to true ($g_?^+ \wedge_{[\Gamma]} g_t^+$) or when the test $g_?$ evaluates to false and the false branch g_f evaluates to true ($g_?^- \wedge_{[\Gamma]} g_f^+$). The two lists of environments are then concatenated, representing a disjunction. The if-then-else returns false in two cases: when the test $g_?$ returns true and the true branch g_t returns false, or when the test $g_?$ returns false and the false branch g_f returns false.

The last type of guard is a type assertion of the form v is T. In this case, the guard returns true if v has type T ($[v \mapsto T]$) and it returns false if v does not have type T ($[v \mapsto \neg T]$).

6.4.3 Clauses

The type of a clause is a combination of the pattern and guard: while the pattern determines the type of terms the clause will accept, the guard may assert type constraints on variables which occur in the pattern.

It is easiest to start with the guard: $\mathcal{G}\,[\![g]\!]$ returns two lists of typing environments: those in which the guard evaluates to true, and those in which the guard evaluates to false. The clause will only match a term if the evaluation of the guard succeeds, so we use the first list of environments. Each environment from the list is then used as an input to $\mathcal{P}\,[\![p]\!]_\Gamma$, each instance of which produces a single type. Afterwards, the union of all of these types is taken: the clause will match when the pattern match succeeds and the guard evaluates to true via *any* control flow branch. The clause's expression e is unused: it does not affect whether or not the clause matches a term.

$$\begin{aligned}\mathcal{C}^* \llbracket c_1, c_2, \ldots, c_{n-1}, c_n \rrbracket \overset{\text{def}}{=} & \ \mathcal{C}\llbracket c_1 \rrbracket \sqcup \\ & (\mathcal{C}\llbracket c_2 \rrbracket \sqcap \neg\mathcal{C}\llbracket c_1 \rrbracket) \sqcup \\ & \ldots \sqcup \\ & (\mathcal{C}\llbracket c_n \rrbracket \sqcap \neg\mathcal{C}\llbracket c_{n-1} \rrbracket \sqcap \ldots \sqcap \neg\mathcal{C}\llbracket c_2 \rrbracket \sqcap \neg\mathcal{C}\llbracket c_1 \rrbracket)\end{aligned}$$

Figure 6.13: Type inference for CoErl clause sequences

6.4.4 Clause Sequences

Clauses typically appear in sequences, where the clauses in the sequence are tried in order until a matching clause (if any) is found. The type of Erlang values accepted by a sequence of clauses is a union of all of the types of the individual clauses: if one clause accepts integers and a second clause accepts booleans, then together the two clauses accept integers and booleans:

```
case X of % accepts integers and booleans
  N when is_integer(N) -> % accepts integers
    do_something();
  B when is_boolean(B) -> % accepts booleans
    do_something_else()
end.
```

As more clauses are added to a `case` or `receive` expression, additional unions are added to the type: if clauses c_1, c_2, and c_3 accept Erlang values of types T_1, T_2, and T_3 respectively, then the combination of those clauses accepts values of type $T_1 \sqcup T_2 \sqcup T_3$.

This technique is suitable for determining the types of values accepted by a sequence of clauses, but it often over-approximates the types accepted by any single clause in a sequence. Consider two clauses where the first accepts values of type S and the second of type T, where $S \leqslant T$:

```
case X of % accepts numbers
  N when is_integer(N) -> % accepts integers
    do_something();
  N when is_number(N) -> % accepts numbers, but not integers
    do_something_else()
```

As clauses are tried in order, any values of type `integer()` will match the first clause, while any *other* numbers (i.e. those which do *not* have type `integer()`) will match the second. Instead of inferring the type $\mathtt{number}()$ for the second clause, a better approximation would be $\mathtt{number}() \sqcap \neg\mathtt{integer}()$: the type of the clause *and not* the type of any preceding clauses.

This approach is followed in the definition of $\mathcal{C}^* [\![\cdot]\!]$ in figure 6.13 on the preceding page, the clause sequence type inference function. The type of a clause sequence is defined as the union of the type of the first clause, the type of the second intersected with the negation of the first clause, and so on.

Chapter 7

Semantic Sub-Typing with BDDs

In section 6.2 on page 97 types are given meaning via a denotational semantics: each type represents the set of values which inhabit it. This model is mainly used to define the sub-typing relation, wherein one type is a sub-type of another if all values of the first type are also values of the second:

$$S \leqslant T \iff [\![S]\!] \subseteq [\![T]\!]$$

The denotational semantics can also be used to derive theorems about types, such as identity elements for union and intersection, and the involution of negation:

$$\begin{aligned} [\![\mathrm{term}() \sqcap T]\!] &= [\![\mathrm{term}()]\!] \cap [\![T]\!] = [\![T]\!] \\ [\![(\neg\mathrm{term}()) \sqcup T]\!] &= [\![\neg\mathrm{term}()]\!] \cup [\![T]\!] = [\![T]\!] \\ [\![\neg\neg T]\!] &= [\![T]\!] \end{aligned}$$

Unfortunately, this abstract model was not immediately useful for automatically deciding the sub-typing relation in a programming language such as Erlang. Instead, we adapted an existing algorithm (Pearce 2013) which checks the sub-typing relation by canonicalising the syntax of a given type and performing an equality check (section 6.3 on page 100). The algorithm is a step in the right direction: it dispenses with the semantic model entirely, but relies on it to prove the correctness of all rewrites and transformations.

Indeed, the algorithm can be implemented in a programming language such as Erlang: sub-typing and intersection functions for positive type atoms can be written using pattern matching, and the canonicalisation function could be written recursively, terminating once no more rewrites are possible. Again, however, there is a problem: the algorithm can require exponential time in the worst case due to the use of a disjunctive normal form in an intermediate step.

Table 7.1: Comparing the efficiency of different representations for boolean formulae

		test for		boolean operations		
Representation	compact?	satisfiability	validity	$\wedge$	$\vee$	$\neg$
Propositional formulas	often	hard	hard	easy	easy	easy
Formulas in DNF	sometimes	easy	hard	hard	easy	hard
Formulas in CNF	sometimes	hard	easy	easy	hard	hard
Ordered truth tables	never	hard	hard	hard	hard	hard
ROBBDs	often	easy	easy	medium	easy	easy

This chapter offers an alternative sub-typing algorithm based on BDDs: a graph-like data structure often used to represent boolean decision procedures. Specifically, we will use *Reduced Ordered Binary Decision Diagrams* (ROBBDs) to represent types, which are BDDs with restrictions on the ordering and duplication of nodes in the graph. ROBBDs offer several advantages when compared to other representations of boolean formulas, as shown by table 7.1: they are often more compact, easier to check for satisfiability, and calculating the conjunction, disjunction, and negation of BDDs[1] is relatively straightforward (Huth and Ryan 2004, ch. 6).

Although our types are not propositional formulas in the strictest sense, types do form a boolean algebra in the denotational semantics: union is disjunction, and intersection is conjunction.

The main contribution of this chapter is a novel approach for deciding a sub-typing relation using BDDs, specifically ROBBDs. This requires modifying the standard ROBBD construction algorithms because BDDs are typically used to represent boolean formulae consisting of *independent* variables, but parts of a type may be related to each other via the sub-typing relation. We will modify the algorithms to ensure that all BDD encodings of types are canonical; this allows us to determine whether two types are semantically equivalent (with respect to the denotational semantics in section 6.2) by checking whether two BDDs are structurally identical. In addition, by using a multi-rooted graph based approach structural equality checks are further reduced to checking whether two pointers are equal.

This novel method of deciding a sub-typing relation does not require normalising or rewriting any type. Instead, types are directly encoded as BDDs by the algorithms presented herein, which gradually construct a single graph which represents one or more types.

Overview This chapter relies on definitions and standard algorithms for constructing BDDs, OBDDs, and ROBBDs, which are located in section 2.3 on page 27.

[1]The terms BDD and ROBBD are often used interchangeably: most implementations of BDD are actually ROBBD due to the speed and size advantages they offer.

$$v \in T \overset{\text{def}}{=} \begin{cases} \mathbf{1} & \text{if } T = \mathit{term}() \\ \texttt{erlang:is\_T}(v) = \texttt{'true'} & \text{if } T \text{ is primitive} \\ v = l & \text{if } T \text{ is a literal } l \\ (v \in S) \vee (v \in U) & \text{if } T = S \sqcup U \\ (v \in S) \wedge (v \in U) & \text{if } T = S \sqcap U \\ \neg(v \in S) & \text{if } T = \neg S \\ v = [\,] & \text{if } T = [\,] \\ (v \in T_h) \wedge (v \in T_t) & \text{if } T = [\, T_h \mid T_t \,] \\ \texttt{erlang:is\_list}(v) = \texttt{'true'} & \text{if } T = \mathit{list}() \\ v = \{\,\} & \text{if } T = \{\,\} \\ \bigwedge_{i=1}^{n} \mathit{elem}(n, v) \in T_n & \text{if } T = \{T_1, T_1, \ldots, T_n\} \\ \texttt{erlang:is\_tuple}(v) = \texttt{'true'} & \text{if } T = \mathit{tuple}() \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Figure 7.1: Decidable type membership

We start by showing how the types from chapter 6 can be represented as BDDs, starting with primitive types, then union, intersection, and negation types, followed by compound types (section 7.1). Afterwards, we modify the standard ROBBD construction algorithms from section 2.3 so that the BDDs which represent types are canonicalised as they are constructed, using the sub-typing algorithm for positive type atoms from chapter 6. This canonicalisation also eliminates semantically redundant nodes from the graph during construction, ensuring each BDD is as small as possible (section 7.2). Finally, we show how sub-typing can be decided by constructing a BDD and checking whether it is satisfiable, which is straightforward using the *Multi-Rooted Directed Acyclic Graph* (MRDAG) approach used in this chapter (section 7.3).

7.1 Representing Types as BDDs

The denotational semantics for types associates the syntax of a type with the set of values which inhabit it (section 6.2). Furthermore, determining whether a given Erlang value is a member of a type is decidable: membership of primitive types can be decided by type test BIFs, compound type membership by pattern matching, and union, intersection, and negation type membership can be decided using the boolean operators for disjunction, conjunction, and negation (figure 7.1).

As type membership can be represented by a boolean decision procedure, it stands to reason that type membership can also be represented as some kind of BDD. While the leaves of the BDDs will continue to hold boolean values, the nodes will contain *positive type atoms*, i.e. primitive or compound types without any unions, intersections or negations. All type operators will be handled by the `If-Then-Else` algorithm instead, as each of them can be represented with an if-then-else:

$$
\begin{aligned}
v \in [\![S \sqcup T]\!] &\implies (v \in [\![S]\!]) \vee (v \in [\![T]\!]) \implies \textbf{if } v \in [\![S]\!] \textbf{ then } \mathbf{1} \textbf{ else } v \in [\![T]\!] \\
v \in [\![S \sqcap T]\!] &\implies (v \in [\![S]\!]) \wedge (v \in [\![T]\!]) \implies \textbf{if } v \in [\![S]\!] \textbf{ then } v \in [\![T]\!] \textbf{ else } \mathbf{0} \\
v \in [\![\neg T]\!] &\implies \neg(v \in [\![T]\!]) \qquad\qquad \implies \textbf{if } v \in [\![T]\!] \textbf{ then } \mathbf{0} \textbf{ else } \mathbf{1}
\end{aligned}
$$

The simplest type to represent with this approach is $\mathtt{term}()$: as all values are members of this type, it can be represented by the leaf #**1**:

$$v \in [\![\mathtt{term}()]\!] \implies \mathbf{1} \implies \#\mathbf{1}$$

The rest of the type system will be handled incrementally, starting with primitive types, then type operators, and finally compound types.

7.1.1 Primitive Types

Primitive types (T^p in section 6.1 on page 95) are relatively straightforward to represent as BDDs as there are no compound type constructors, conjunctions, disjunctions, or negations. Furthermore, as all primitive types are also positive type atoms, we can place the primitive type directly into a node in the BDD.

Therefore, to obtain a pointer to a BDD representing $v \in T^p$ in the MRDAG G, we can use `Find-Or-Create` (algorithm 2 on page 33):

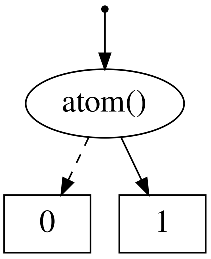
$$v \in [\![T^p]\!] \implies \texttt{Find-Or-Create}(T^p, \#\mathbf{1}, \#\mathbf{0})$$

As `Find-Or-Create` only creates a new node if an equivalent one doesn't exist in G, and both the $\mathtt{hi}$ edge and $\mathtt{lo}$ edge point to leaves, the pointer returned by `Find-Or-Create` will represent a reduced and ordered BDD for deciding membership of T^p. Figure 7.2 on the following page shows two examples of how the primitive types $\mathtt{atom}()$ (figure 7.2a) and $\mathtt{integer}()$ (figure 7.2b) are represented as BDDs.

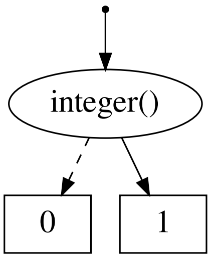
For the sake of completeness, `Add-Primitive` defined in algorithm 5 on the next page will be used to create ROBBDs for primitive types.

7.1.2 Union, Intersection, and Negations

At the beginning of this section on page 120 we noted that union, intersection, and negation types correspond to logical disjunction, conjunction, and negation when deciding type membership.



(a) atom()



(b) integer()

Figure 7.2: Example BDD representations of primitive types

```
Function Add-Primitive($T^p$, G)
    Data: A primitive type $T^p$
    Data: An MRDAG G
    Result: A pointer to a BDD in the MRDAG representing a test for type $T^p$
    $r \leftarrow$ Find-Or-Create ($T^p$, ptr(#1), ptr(#0), G)
    return $r$
end
```

Algorithm 5: Add-Primitive for adding a primitive type to an MRDAG based ROBDD

Furthermore, as each of these logical operations can be represented by an equivalent if-then-else expression, we can take advantage of the `If-Then-Else` algorithm discussed in section 2.3.3.

Assuming that S_p and T_p are pointers to the roots of BDDs in an MRDAG representing decision procedures for $v \in [\![S]\!]$ and $v \in [\![T]\!]$ respectively, then we can create a BDD for any type operator:

$$\begin{aligned}
v \in [\![S \sqcup T]\!] &\implies \textbf{if } v \in [\![S]\!] \textbf{ then 1 else } v \in [\![T]\!] \implies \texttt{If-Then-Else}(S_p, \#\mathbf{1}, T_p) \\
v \in [\![S \sqcap T]\!] &\implies \textbf{if } v \in [\![S]\!] \textbf{ then } v \in [\![T]\!] \textbf{ else 0} \implies \texttt{If-Then-Else}(S_p, T_p, \#\mathbf{0}) \\
v \in [\![\neg T]\!] &\implies \textbf{if } v \in [\![T]\!] \textbf{ then 0 else 1} \qquad \implies \texttt{If-Then-Else}(S_t, \#\mathbf{0}, \#\mathbf{1})
\end{aligned}$$

As an example, we convert the type $(S \sqcap T) \sqcup U$ into a BDD by repeated application of the above rules:

$$\begin{aligned}
& v \in [\![(S \sqcap T) \sqcup U]\!] \\
\implies & \textbf{if } v \in [\![S \sqcap T]\!] \textbf{ then 1 else } v \in [\![U]\!] \\
\implies & \textbf{if } (\textbf{if } v \in [\![S]\!] \textbf{ then } v \in [\![T]\!] \textbf{ else 0}) \textbf{ then 1 else } v \in [\![U]\!] \\
\implies & \textbf{if } (\textbf{if } S_p \textbf{ then } T_p \textbf{ else 0}) \textbf{ then 1 else } U_p \\
\implies & \textbf{if } \texttt{If-Then-Else}(S_p, T_p, \#\mathbf{0}) \textbf{ then 1 else } U_p \\
\implies & \texttt{If-Then-Else}(\texttt{If-Then-Else}(S_p\ T_p, \#\mathbf{0}),\ \#\mathbf{1},\ U_p)
\end{aligned}$$

In general, any $\sqcup$, $\sqcap$, and $\neg$ type can be converted into BDDs by first converting it into if-then-else form and then applying the `If-Then-Else` algorithm.

7.1.3 Compound Types

The `If-Then-Else` algorithm combines *existing* BDDs into new ones, working from the bottom up. Therefore, to convert the type $(S \sqcap T) \sqcup U$ into a BDD, we must start at the deepest part of the type and work our way towards the top:

1. Create a BDD for types S and T.
2. Create BDDs for types $S \sqcap T$ (using the results from step 1) and U.
3. Create a BDD for type $(S \sqcap T) \sqcup U$ using the result from step 2.

For compound types which form positive type atoms this is not an issue: types such as $[\,\mathtt{atom}() \mid \mathtt{integer}()\,]$ and $\{\texttt{'req'}, \mathtt{integer}(), \ldots, \mathtt{integer}()\}$ do not contain any type operators and can therefore be placed in nodes. The issue arises when types *do not* form positive type atoms: as $[\,\mathtt{atom}() \mid \neg\mathtt{integer}()\,]$ contains a type operator it is not a positive atom and cannot be placed in a BDD node.

Instead, we will take advantage of the fact that any compound type can be split into an intersection where each member of the intersection effectively only checks the type of a specific element by setting the types of all other elements to $\mathtt{term}()$:

$$\begin{aligned}
[\![[\,T_h \mid T_t\,]]\!] &\Longleftrightarrow [\![[\,T_h \mid \mathtt{term}()\,] \sqcap [\,\mathtt{term}() \mid T_t\,]]\!] \\
[\![\{T_1, T_2, \ldots, T_n\}]\!] &\Longleftrightarrow [\![\{T_1, \mathtt{term}(), \ldots, \mathtt{term}()\} \sqcap \{\mathtt{term}(), T_2, \ldots, \mathtt{term}()\} \sqcap \ldots \\
&\qquad \sqcap \{\mathtt{term}(), \mathtt{term}(), \ldots, T_n\}]\!]
\end{aligned}$$

which, going back to our example, means:

$$\begin{aligned}
[\,\mathtt{atom}() \mid \mathtt{integer}()\,] &\Longleftrightarrow [\,\mathtt{atom}() \mid \mathtt{term}()\,] \sqcap [\,\mathtt{term}() \mid \mathtt{integer}()\,] \\
[\,\mathtt{atom}() \mid \neg\mathtt{integer}()\,] &\Longleftrightarrow [\,\mathtt{atom}() \mid \mathtt{term}()\,] \sqcap [\,\mathtt{term}() \mid \neg\mathtt{integer}()\,]
\end{aligned}$$

This concept lies at the heart of the `Add-Type` in figure 7.3 on the following page, which converts any type into a BDD which decides membership of that type. The function relies on a context function `f` which represents a hole into which a primitive type, literal, or empty compound type constructor can be placed to generate a positive type atom. For example, the context function $\lambda x.\, [\,x \mid \mathtt{term}()\,]$ produces the positive type atom $[\,\mathtt{boolean}() \mid \mathtt{term}()\,]$ when

$$\texttt{Add-Type}(T, f) \overset{\text{def}}{=} \begin{cases} \texttt{Find-Or-Create}(f(T),\ \#\mathbf{1},\ \#\mathbf{0}) & \text{if } T = term() \\ \texttt{Find-Or-Create}(f(T),\ \#\mathbf{1},\ \#\mathbf{0}) & \text{if } T \text{ is primitive} \\ \texttt{Find-Or-Create}(f(T),\ \#\mathbf{1},\ \#\mathbf{0}) & \text{if } T \text{ is a literal} \\ \texttt{Add-Type}(S, f) \vee^G \texttt{Add-Type}(U, f) & \text{if } T = S \sqcup U \\ \texttt{Add-Type}(S, f) \wedge^G \texttt{Add-Type}(U, f) & \text{if } T = S \sqcap U \\ \neg^G \texttt{Add-Type}(S, f) & \text{if } T = \neg S \\ \texttt{Find-Or-Create}(f([\,]),\ \#\mathbf{1},\ \#\mathbf{0}) & \text{if } T = [\,] \\ \texttt{Add-Type}(T_h, f_h) \wedge^G \texttt{Add-Type}(T_t, f_t) & \text{if } T = [\, T_h \mid T_t \,] \\ \quad \textbf{where } f_h = \lambda x. [\, f(x) \mid term() \,] \text{ and} & \\ \qquad\qquad f_t = \lambda x. [\, term() \mid x \,] & \\ \texttt{Find-Or-Create}(f(\{\,\}),\ \#\mathbf{1},\ \#\mathbf{0}) & \text{if } T = \{\,\} \\ \texttt{Add-Type}(T_1, f_1) \wedge^G \ldots \wedge^G \texttt{Add-Type}(T_n, f_n) & \text{if } T = \{T_1, T_2, \ldots, T_n\} \\ \quad \textbf{where } f_1 = \lambda x. \{f(x), term(), \ldots, term()\} & \\ \qquad\qquad f_2 = \lambda x. \{term(), f(x), \ldots, term()\} & \\ \qquad\qquad \vdots & \\ \qquad\qquad f_n = \lambda x. \{term(), term(), \ldots, f(x)\} & \end{cases}$$

$$A \vee^G B \overset{\text{def}}{=} \texttt{If-Then-Else}(A, \#\mathbf{1}, B)$$
$$A \wedge^G B \overset{\text{def}}{=} \texttt{If-Then-Else}(A, B, \#\mathbf{0})$$
$$\neg^G A \overset{\text{def}}{=} \texttt{If-Then-Else}(A, \#\mathbf{0}, \#\mathbf{1})$$

Figure 7.3: Add-Type function for converting a type to an MRDAG based BDD

applied to the primitive type $boolean()$. This context function allows us to descend through the type while keeping track of which part of a compound type (if any) we are currently inside.

The `Add-Type` operates on an implicit global MRDAG, similar to how `If-Then-Else` and `Find-Or-Create` behave. To convert a type T into a BDD representing a decision procedure for membership of type T, we call $\texttt{Add-Type}(T, \lambda x.x)$, where the identity function represents a "top level" hole. For $term()$, primitive types, and literals, we attempt to create a new node in the graph which checks for membership of the primitive type $f(T)$. We cannot simply return the leaf # for type $term()$, however, as we might not be at the top-level context: $term() \neq [\, term() \mid term() \,]$, for example.

We then have the type operators: $\sqcup$, $\sqcap$, and $\neg$. In each case we make a recursive call to `Add-Type` using the relevant operands *and the same context function*, combining the results using an appropriate call to the `If-Then-Else` algorithm. This utilises a theorem from section 6.3 which states that any operator can be “lifted” out of a compound type:

$$\llbracket [\, S_h \sqcup T_h \mid T_t \,] \rrbracket \iff \llbracket [\, S_h \mid T_t \,] \sqcup [\, S_h \mid T_t \,] \rrbracket$$

$$\llbracket [\, S_h \sqcup T_h \mid T_t \,] \rrbracket \iff \llbracket [\, S_h \mid T_t \,] \sqcap [\, T_h \mid T_t \,] \rrbracket$$

$$\llbracket [\, \neg T_h \mid T_t \,] \rrbracket \iff \llbracket [\, \mathit{term}() \mid T_t \,] \sqcap \neg [\, T_h \mid \mathit{term}() \,] \rrbracket$$

When a non-empty compound type constructor is encountered by `Add-Type` it first splits the constructor into an conjunction, where each member of the conjunction checks the type of a single element by wrapping the existing hole with a compound type constructor. For example, with the type $[\, \mathit{atom}() \mid \mathit{boolean}() \sqcup \mathit{integer}() \,]$:

$$
\begin{aligned}
&\quad v \in \llbracket [\, \mathit{atom}() \mid \mathit{boolean}() \sqcup \mathit{integer}() \,] \rrbracket \\
&\Longrightarrow \texttt{Add-Type}([\, \mathit{atom}() \mid \mathit{boolean}() \sqcup \mathit{integer}() \,],\ \lambda x.x) \\
&\Longrightarrow \texttt{Add-Type}(\mathit{atom}(), \lambda x.\, [\, x \mid \mathit{term}() \,]) \ \wedge^{G}\ \texttt{Add-Type}(\mathit{boolean}() \sqcup \mathit{integer}(), \lambda x.\, [\, \mathit{term}() \mid x \,]) \\
&\Longrightarrow \texttt{Add-Type}(\mathit{atom}(), \lambda x.\, [\, x \mid \mathit{term}() \,]) \wedge^{G} \\
&\qquad (\texttt{Add-Type}(\mathit{boolean}(), \lambda x.\, [\, \mathit{term}() \mid x \,]) \ \vee^{G}\ \texttt{Add-Type}(\mathit{integer}(), \lambda x.\, [\, \mathit{term}() \mid x \,])) \\
&\Longrightarrow \texttt{Find-Or-Create}([\, \mathit{atom}() \mid \mathit{term}() \,], \#\mathbf{1}, \#\mathbf{0}) \wedge^{G} \\
&\qquad (\texttt{Find-Or-Create}([\, \mathit{term}() \mid \mathit{boolean}() \,], \#\mathbf{1}, \#\mathbf{0}) \vee^{G} \\
&\qquad\ \texttt{Find-Or-Create}([\, \mathit{term}() \mid \mathit{integer}() \,], \#\mathbf{1}, \#\mathbf{0})) \\
&\Longrightarrow \texttt{If-Then-Else}(\texttt{Find-Or-Create}([\, \mathit{atom}() \mid \mathit{term}() \,], \#\mathbf{1}, \#\mathbf{0}), \\
&\qquad \texttt{If-Then-Else}(\texttt{Find-Or-Create}([\, \mathit{term}() \mid \mathit{boolean}() \,], \#\mathbf{1}, \#\mathbf{0}), \\
&\qquad\qquad\qquad \#\mathbf{1}, \texttt{Find-Or-Create}([\, \mathit{term}() \mid \mathit{integer}() \,], \#\mathbf{1}, \#\mathbf{0})), \#\mathbf{0})
\end{aligned}
$$

which is equivalent to first converting the type into if-then-else syntax and then applying the

`If-Then-Else` and `Find-Or-Create` algorithms:

$$
\begin{aligned}
& v \in [\![[\,\mathtt{atom}() \mid \mathtt{boolean}() \sqcup \mathtt{integer}()\,]]\!] \\
\Longrightarrow\ & v \in [\![[\,\mathtt{atom}() \mid \mathtt{term}()\,] \sqcap [\,\mathtt{term}() \mid \mathtt{boolean}() \sqcup \mathtt{integer}()\,]]\!] \\
\Longrightarrow\ & v \in [\![[\,\mathtt{atom}() \mid \mathtt{term}()\,] \sqcap [\,\mathtt{term}() \mid \mathtt{boolean}() \sqcup \mathtt{integer}()\,]]\!] \\
\Longrightarrow\ & v \in [\![[\,\mathtt{atom}() \mid \mathtt{term}()\,] \sqcap ([\,\mathtt{term}() \mid \mathtt{boolean}()\,] \sqcup [\,\mathtt{term}() \mid \mathtt{integer}()\,])]\!] \\
\Longrightarrow\ & \textbf{if } v \in [\![[\,\mathtt{atom}() \mid \mathtt{term}()\,]]\!] \textbf{ then } v \in [\![[\,\mathtt{term}() \mid \mathtt{boolean}()\,] \sqcup [\,\mathtt{term}() \mid \mathtt{integer}()\,]]\!] \textbf{ else } \mathbf{0} \\
\Longrightarrow\ & \textbf{if } v \in [\![[\,\mathtt{atom}() \mid \mathtt{term}()\,]]\!] \textbf{ then} \\
& \quad (\textbf{if } v \in [\![[\,\mathtt{term}() \mid \mathtt{boolean}()\,]]\!] \textbf{ then } \mathbf{1} \textbf{ else } v \in [\![[\,\mathtt{term}() \mid \mathtt{integer}()\,]]\!]) \textbf{ else } \mathbf{0} \\
\Longrightarrow\ & \textbf{if } \texttt{Find-Or-Create}([\,\mathtt{atom}() \mid \mathtt{term}()\,], \#\mathbf{1}, \#\mathbf{0}) \textbf{ then} \\
& \quad (\textbf{if } \texttt{Find-Or-Create}([\,\mathtt{term}() \mid \mathtt{boolean}()\,], \#\mathbf{1}, \#\mathbf{0}) \textbf{ then} \\
& \qquad \mathbf{1} \textbf{ else } \texttt{Find-Or-Create}([\,\mathtt{term}() \mid \mathtt{integer}()\,], \#\mathbf{1}, \#\mathbf{0})) \textbf{ else } \mathbf{0} \\
\Longrightarrow\ & \texttt{If-Then-Else}(\texttt{Find-Or-Create}([\,\mathtt{atom}() \mid \mathtt{term}()\,], \#\mathbf{1}, \#\mathbf{0}), \\
& \quad \texttt{If-Then-Else}(\texttt{Find-Or-Create}([\,\mathtt{term}() \mid \mathtt{boolean}()\,], \#\mathbf{1}, \#\mathbf{0}), \\
& \qquad\qquad \#\mathbf{1}, \texttt{Find-Or-Create}([\,\mathtt{term}() \mid \mathtt{integer}()\,], \#\mathbf{1}, \#\mathbf{0})), \#\mathbf{0})
\end{aligned}
$$

This will create a BDD similar to the one shown in figure 7.4 on the next page. At the root, membership of the positive type atom $[\,\mathtt{atom}() \mid \mathtt{term}()\,]$ is checked, essentially checking that a given term is a cons cell whose first element has type $\mathtt{atom}()$. The other two nodes represent a disjunction, checking that the second element of the cons cell is either a member of $\mathtt{boolean}()$ or $\mathtt{integer}()$.

7.2 Canonicalisation

We convert the question of type membership ($v \in [\![T]\!]$) into a BDD by converting types into if-then-else expressions which can be handled by the `If-Then-Else` algorithm. Each type is "flattened" as it is passed to `If-Then-Else` so that each node in the graph contains a *positive type atom*. By rewriting types into this form we move all union, intersection, and negation operators outside of compound type constructors. This allows us to build a canonical BDD for each type consisting of unions, intersections, and negations of positive type atoms up to some (as yet undefined) ordering on types. We will only need to define an ordering for positive type atoms because nodes in our BDDs will only ever contain positive type atoms.

The `If-Then-Else` algorithm however assumes that the value contained in each node is independent of all others on the same path. That is to say that the BDDs created by `If-Then-Else`

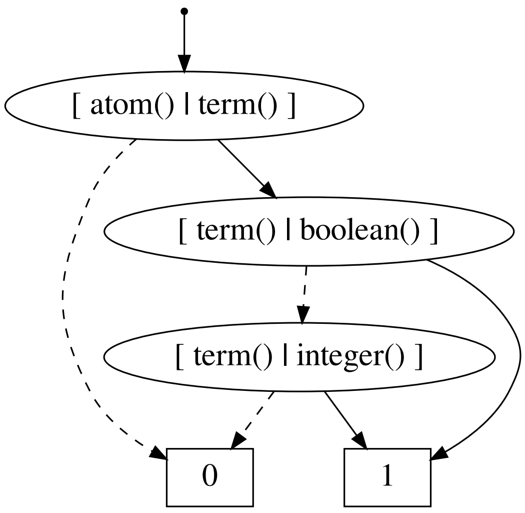


Figure 7.4: Type [$\mathtt{atom}()$ | $\mathtt{boolean}() \sqcup \mathtt{integer}()$] in BDD form

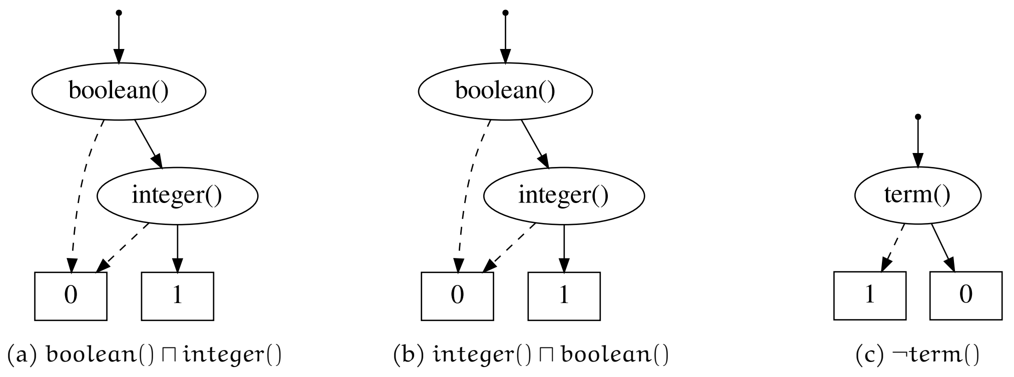
are canonical *up to the boolean algebra they represent*. When converting boolean formulas to BDDs this is perfectly acceptable: we can encode boolean operations such as conjunction, disjunction, negation, implication, and exclusive or using if-then-else expressions. In all of these situations the variables are distinct: each variable only has one name and any dependencies between variables are expressed via if-then-else expressions.

With type based BDDs however the "variables" are *not* independent. When we convert a type to a BDD we imagine that we are checking the type of an imaginary variable v:

$$\begin{aligned} & v \in [\![\mathtt{boolean}() \sqcap \mathtt{integer}()]\!] \\ = {} & (v \in [\![\mathtt{boolean}()]\!]) \wedge (v \in [\![\mathtt{integer}()]\!]) \\ = {} & \textbf{if } v \in [\![\mathtt{boolean}()]\!] \textbf{ then } v \in [\![\mathtt{integer}]\!] \textbf{ else } \mathbf{0} \end{aligned}$$

This imaginary variable hints at the dependence between type atoms in nodes: if the entire BDD represents a type test on variable v, then each node on a path through the BDD provides information about the type of v. Essentially the `If-Then-Else` algorithm is designed to combine if-then-else expressions which reason about the values of many distinct boolean variables, while we are using BDDs to reason about the type of a single Erlang value.

Consider the ROBBDs in figure 7.5 on the following page which all use the same ordering on types and which all represent a type semantically equivalent to the empty set. Figure 7.5a and figure 7.5b are structurally identical due to the commutativity of the $\sqcap$ operator. Figure 7.5c is structurally different however, as it represents a semantically different *boolean formula*. For example, if we substitute types with variable names (e.g. X is **1** iff v is a $\mathtt{boolean}()$, and Y is true



(a) $\mathtt{boolean}() \sqcap \mathtt{integer}()$ (b) $\mathtt{integer}() \sqcap \mathtt{boolean}()$ (c) $\neg\mathtt{term}()$

Figure 7.5: Non canonicalised representations of semantically equivalent types

iff v is an integer()) we can see that the two "formulas" are semantically different:

$$\mathtt{boolean}() \sqcap \mathtt{integer}() \Longrightarrow X \wedge Y$$
$$\mathtt{integer}() \sqcap \mathtt{boolean}() \Longrightarrow Y \wedge X$$
$$\neg\mathtt{term}() \Longrightarrow Z$$

In order to canonicalise type BDDs we must therefore consider the relationship between types.

The BDD construction algorithm will be modified in two places: we will add a special case for the Find-Or-Create algorithm handle the term() type, and we will use sub-typing and intersection of positive type atoms in Restrict.

7.2.1 Modified Find-Or-Create Algorithm

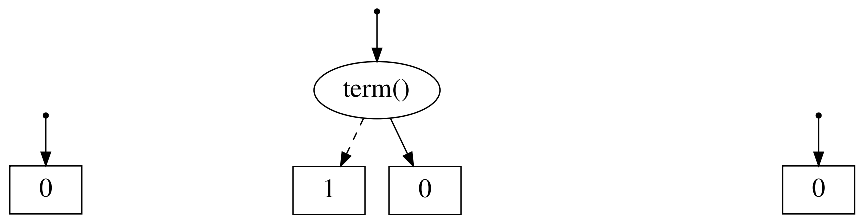
The Find-Or-Create algorithm (algorithm 2) is the only place where nodes can be inserted into the graph, and only when doing so will not create any duplicates. Unfortunately, it inserts a node regardless of the value contained within it, including nodes for term(). We however know that *every value* is a member of type term() as it is the top type, so any membership of the test $v \in [\![\mathtt{term}()]\!]$ will always succeed.

Therefore we will slightly modify the Find-Or-Create algorithm to deal with the term() type explicitly. Algorithm 6 on the next page shows the new Find-Or-Create-Ty algorithm which is different to Find-Or-Create in one place: on line 0 we check whether the variable v is term(), and return the hi edge if so:

$$\textbf{if } v \in [\![\mathtt{term}()]\!] \textbf{ then } S \textbf{ else } T \Longleftrightarrow S$$
$$\mathtt{Find\text{-}Or\text{-}Create\text{-}Ty}(\mathtt{term}(), S, T) \Longleftrightarrow S$$

```
Function Find-Or-Create($v$, $\mathtt{hi}$, $\mathtt{lo}$, $G$)
    Data: A node $\langle v\ ?\ \mathtt{hi} : \mathtt{lo}\rangle$ to add to the MRDAG $G$
    Data: The existing MRDAG $G$
    Result: A pointer to a node equivalent to $\langle v\ ?\ \mathtt{hi} : \mathtt{lo}\rangle$
TERM    if $v$ is $\mathtt{term}()$ then
1           return $\mathtt{hi}$
    if there is a node $n \in G$ such that $n = \langle v\ ?\ \mathtt{hi} : \mathtt{lo}\rangle$ then
        return pointer to $n$
    else
        insert $\langle v\ ?\ \mathtt{hi} : \mathtt{lo}\rangle$ into $G$
        return pointer to the inserted node
end
```

Algorithm 6: Find-Or-Create-Ty algorithm for Type ROBDDs



(a) #**0** (b) `Find-Or-Create`($\mathtt{term}()$, #**0**, #**1**) (c) `Find-Or-Create-Ty`($\mathtt{term}()$, #**0**, #**1**)

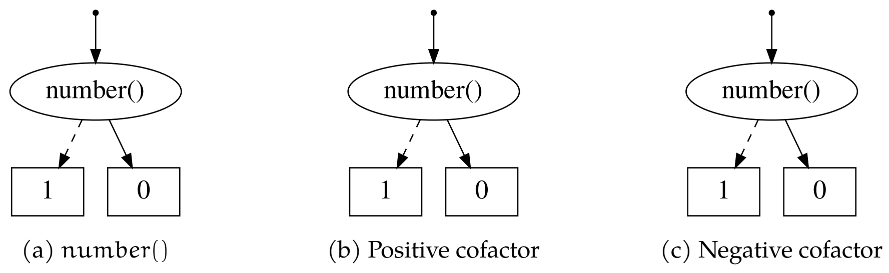
Figure 7.6: Canonicalised representations of semantically equivalent types

This change allows us to canonicalise occurrences of $\mathtt{term}()$ within types by always returning the $\mathtt{hi}$ edge. The BDDs in figure 7.6 show how this canonicalisation occurs. In figure 7.6a we have the constant BDD **0** which represents the empty type $\neg\mathtt{term}()$, then we have used two different algorithms to insert the node $(\mathtt{term}(), \#\mathbf{0}, \#\mathbf{1})$ (representing the same type $\neg\mathtt{term}()$) into the graph: `Find-Or-Create` (figure 7.6b) and `Find-Or-Create-Ty` (figure 7.6c). In the first case the redundant $\mathtt{term}()$ node was created as the type $\mathtt{term}()$ is treated like any other variable, but in the second case the $\mathtt{hi}$ edge was returned, avoiding the redundant node (which points to #**0**).

7.2.2 Modified Restrict Algorithm

The current `Restrict` algorithm operates by considering the value of the node at the root of a BDD. As the graph remains fully reduced and ordered at all times we know that the root node will either be the variable being eliminated, or it will be another greater variable (according to the ordering).

This is extremely useful when considering *independent* variables: if the variable being restricted is at the root, we can create the positive and negative cofactors using the $\mathtt{hi}$ and $\mathtt{lo}$ edges of



(a) $\mathtt{number}()$ (b) Positive cofactor (c) Negative cofactor

Figure 7.7: Incorrect cofactors of $\mathtt{number}()$ restricted w.r.t. $\mathtt{integer}()$

the root, and in all other cases we return the root node itself (algorithm 4).

The BDDs in figure 7.7 show that this method is inadequate for calculating the cofactors of types, however. The first BDD (figure 7.7a) represents the type $\mathtt{number}()$ while the second and third show the positive and negative cofactors w.r.t. the type $\mathtt{integer}()$ (figures 7.7b and 7.7c). When creating the positive cofactor we are making the assumption that $v \in [\![\mathtt{integer}()]\!]$, and in the negative cofactor we assume that $v \notin [\![\mathtt{integer}()]\!]$. If we consider the positive cofactor as an if-then-else expression:

$$\textbf{if } v \in [\![\mathtt{number}()]\!] \textbf{ then 1 else 0}$$

then we start to see an issue when we wrap it in an if-then-else which asserts that $v \in [\![\mathtt{integer}()]\!]$ in the true branch::

$$\textbf{if } v \in [\![\mathtt{integer}()]\!] \textbf{ then if } v \in [\![\mathtt{number}()]\!] \textbf{ then 1 else 0 else 0}$$

As the type assertion $v \in [\![\mathtt{number}()]\!]$ *must* be true if $v \in [\![\mathtt{integer}()]\!]$ as $(\mathtt{integer}() \leqslant \mathtt{number}())$, we should simplify away the type assertion in the true branch of the if-then-else:

$$\begin{aligned} &\textbf{if } v \in [\![\mathtt{integer}()]\!] \textbf{ then if } v \in [\![\mathtt{number}()]\!] \textbf{ then 1 else 0 else 0} \\ =\ &\textbf{if } v \in [\![\mathtt{integer}()]\!] \textbf{ then if 1 then 1 else 0 else 0} && \text{as } \mathtt{integer}() \leqslant \mathtt{number}() \\ =\ &\textbf{if } v \in [\![\mathtt{integer}()]\!] \textbf{ then 1 else 0} \end{aligned}$$

Note that the *negative* cofactor is unchanged: although we know that $v \notin [\![\mathtt{integer}()]\!]$ (contrary to the positive case) and $\mathtt{integer}() \leqslant \mathtt{number}()$, we also know that $\mathtt{number}() \sqcap \neg \mathtt{integer}() \neq \emptyset$, i.e. there are *some* values which are numbers that are not integers. As this type is inhabited by some (but not *all*) values, we cannot eliminate the $\mathtt{number}()$ node in the negative cofactor as it is not redundant. On the contrary, if $\mathtt{integer}()$ were at the root node and we were restricting type $\mathtt{number}()$ we could no longer eliminate the root node when calculating the positive cofactor as $\mathtt{number}() \nleqslant \mathtt{integer}()$.

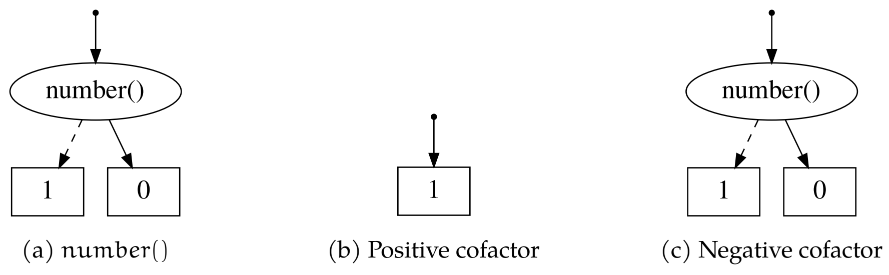


Figure 7.8: Correct cofactors of $\mathtt{number()}$ w.r.t. $\mathtt{integer()}$

In figure 7.7 on the previous page we have three different BDDs: the type $\mathtt{number()}$ and its positive and negative cofactors w.r.t. $\mathtt{integer()}$ as per the current `Restrict` algorithm. As $\mathtt{number()}$ and $\mathtt{integer()}$ are syntactically different, each cofactor is identical to the original. The cofactors here are incorrect however as *type assertions on the same value are not independent*. For example, for the positive cofactor in figure 7.7b we are assuming that $v \in [\![\mathtt{integer()}]\!]$. Therefore, the type test $v \in [\![\mathtt{number()}]\!]$ will always return true and the correct positive cofactor would be the $\mathtt{hi}$ edge of the root node: #**1**. Unfortunately we cannot perform the same optimisation on the negative cofactor (where we assert that $v \notin [\![\mathtt{integer()}]\!]$) as there are other members of $\mathtt{number()}$ which are not also members of $\mathtt{integer()}$.

The first requirement of creating a canonical ROBBD is the ordering. We will use the ordering from figure 7.9 on the following page. This relation has two important properties:

- for all types S and T such that $S \leqslant T$, either $S = T$ or $S < T$.
- compound types are ordered element-wise.

The first property allows us to optimise the specialised `restrict` algorithm we will use to canonicalise types, while the second property ensures that cons cells and tuples are ordered such that all type tests where the non-$\mathtt{term()}$ type is in the same position occur adjacent to oneanother in the graph. This last point is subtle: recall that we create type atoms in a way that the type $\{T_1, T_2, \ldots, T_n\}$ is normalised to:

$$\{T_1, \mathtt{term()}, \ldots, \mathtt{term()}\} \sqcap \{\mathtt{term()}, T_2, \ldots, \mathtt{term()}\} \sqcap \{\mathtt{term()}, \mathtt{term()}, \ldots, T_n\}$$

The ordering ensures that the type atoms $\{S_1, \mathtt{term()}, \ldots, \mathtt{term()}\}$, $\{T_1, \mathtt{term()}, \ldots, \mathtt{term()}\}$, and $\{\mathtt{term()}, T_2, \ldots, \mathtt{term()}\}$ are ordered as:

$$\{S_1, \mathtt{term()}, \ldots, \mathtt{term()}\} < \{T_1, \mathtt{term()}, \ldots, \mathtt{term()}\} < \{\mathtt{term()}, T_2, \ldots, \mathtt{term()}\}$$

$$\mathit{boolean}() <^p \mathit{atom}() <^p \mathit{float}() <^p \mathit{integer}() <^p \mathit{number}() <^p \mathit{pid}() <^p \mathit{port}() <^p \mathit{reference}()$$

$$\frac{S < T \qquad T < U}{S < U}\ \textsc{Trans} \qquad \frac{S \neq \mathit{term}()}{S < \mathit{term}()}\ \textsc{Term} \qquad \frac{T_1^p <^p T_2^p}{T_1^p < T_2^p}\ \textsc{Prim} \qquad \frac{L_1 <^l L_2}{L_1 < L_2}\ \textsc{Lit}$$

$$\frac{}{L < T^p}\ \textsc{LitPrim} \qquad \frac{}{[\,] < [\,T_h \mid T_t\,]}\ \textsc{NilCons} \qquad \frac{S_h < T_h}{[\,S_h \mid S_t\,] < [\,T_h \mid T_t\,]}\ \textsc{ConsHd}$$

$$\frac{S_h = T_h \qquad S_t < T_t}{[\,S_h \mid S_t\,] < [\,T_h \mid T_t\,]}\ \textsc{ConsTl} \qquad \frac{}{[\,T_h \mid T_t\,] < \mathit{list}()}\ \textsc{ListCons} \qquad \frac{}{\mathit{list}() < \{\,\}}\ \textsc{ListTuple}$$

$$\frac{}{\{\,\} < \{T_1, T_2, \dots, T_n\}}\ \textsc{TupleNil} \qquad \frac{m < n}{\{S_1, S_2, \dots, S_m\} < \{T_1, T_2, \dots, T_n\}}\ \textsc{TupleSize}$$

$$\frac{S_1 = T_1 \qquad S_2 = T_2 \qquad \dots \qquad S_{m-1} = T_{m-1} \qquad S_m < T_m}{\{S_1, \dots, S_m, \dots, S_n\} < \{T_1, \dots T_m, \dots, T_n\}}\ \textsc{TupleEq}$$

$$\frac{}{\{S_1, S_2, \dots, S_n\} < \mathit{tuple}()}\ \textsc{TupleN}$$

Figure 7.9: Ordering relation for positive type atoms

when $S_1 \leqslant T_1$. This has an effect when calculating sub-typing. For example, although the first and third atoms intersect:

$$\{S_1, \mathit{term}(), \dots, \mathit{term}()\} \sqcap^* \{\mathit{term}(), T_2, \dots, \mathit{term}()\} = \{S_1, T_2, \dots, \mathit{term}()\}$$

the sub-typing relation does not hold:

$$\{S_1, \mathit{term}(), \dots, \mathit{term}()\} \nleqslant \{\mathit{term}(), T_2, \dots, \mathit{term}()\}$$

By contrast, while the first and second atoms intersect, *the sub-typing relation also holds* because the non-$\mathit{term}()$ type appears in the same position (as $S \leqslant T$):

$$\{S_1, \mathit{term}(), \dots, \mathit{term}()\} \leqslant \{T_1, \mathit{term}(), \dots, \mathit{term}()\}$$

Therefore this ordering of types ensures that:

$$\forall S, T.\, S \leqslant T \Longrightarrow \nexists U.\, U \nleqslant T \land S < U < T$$

The final step in canonicalising type ROBBDs is to modify the `restrict` algorithm to eliminate redundant nodes using the sub-typing relation. This modified version shown in algorithm 7

```
Function Restrict-Type(S, p)
    Data: Type S to restrict on
    Data: Pointer p representing an ROBBD in an MRDAG
    Result: Pointers to the positive and negative cofactors of p restricted to type T
            respectively
    $T \leftarrow$ type held in node p
    if $S = T$ then
        return $(\mathtt{hi}(p), \mathtt{lo}(p))$
    end
    else if $S \leqslant T$ then
        return $(\mathtt{hi}(p), p)$
    end
    else if $S \sqcap^* T \neq \neg\mathtt{term}()$ then
        return $(p, p)$
    end
    else
        return $(\mathtt{lo}(p), p)$
    end
end
```

Algorithm 7: Restrict function for MRDAG based Type ROBDDs

operates specifically on types, not boolean variables. As before, if the two types are syntactically equal we bypass the root node p entirely, returning the hi and lo edges. There are three more possible cases to consider:

1. If $S \leqslant T$ we know that once the type test for S succeeds, then the type test for S will succeeds, so we can return the hi edge of T as the positive cofactor. The negative cofactor remains unchanged because the type test for T may still succeed if the type test for S fails (as we know $S \neq T$).

2. If $S \sqcap^* T \neq \neg\mathtt{term}()$ we know that the two types are *not* disjoint and the type test for T may either succeed or fail (but will not *always* succeed as $S \nleqslant T$). We therefore return p as the positive and negative cofactors.

3. If S and T are disjoint (the `else` branch) and we know that the type test for S has succeeded, then we know that the type test for T will always fail, so we return the lo edge as the positive cofactor.

This modified `Restrict` algorithm (now called `Restrict-Type`) canonicalises types *beyond* the boolean algebra they represent because it has knowledge of the relationship between positive type atoms. The remainder of the BDD construction remains unchanged: we do not create duplicate nodes, we operate on a single MRDAG, and we construct all BDDs using if-then-else expressions.

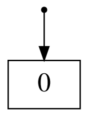


Figure 7.10: BDD for type $[\,\mathit{integer}() \mid \mathit{atom}()\,] \sqcap \neg\,[\,\mathit{number}() \mid \neg\mathit{integer}()\,]$

7.3 Checking the Sub-Typing Relation with BDDs

So far in this chapter we have shown how type membership tests of the form $v \in [\![T]\!]$ can be encoded as ROBBDs. For each *semantically equivalent* types S and T the `Add-Type` algorithm creates structurally identical BDDs: $\texttt{Add-Type}(S) = \texttt{Add-Type}(T)$. Additionally, as these algorithms operate on a single MRDAG and return *pointers* to the roots of BDDs within them, we can check for semantic equality between two types by first converting them to BDDs and then checking whether the pointers returned by `Add-Type` are equal.

To check sub-typing using this approach we recall that sub-typing is defined using the subset relation (definition 6.2.1 on page 99):

$$S \leqslant T \overset{\text{def}}{=} [\![S]\!] \subseteq [\![T]\!]$$

which is equivalent to:

$$[\![S \sqcap \neg T]\!] = \emptyset$$

meaning that S is a sub-type of T if there is no value in S which is not also in T. As a type membership question, this is equivalent to the following statement being unsatisfiable:

$$S \leqslant T \iff \nexists v.\, v \in [\![S \sqcap \neg T]\!]$$

Therefore we can check sub-typing by constructing a BDD: if the BDD for type $S \sqcap \neg T$ is *unsatisfiable*, then $S \leqslant T$:

$$S \leqslant T \iff \texttt{Add-Type}(S \sqcap \neg T) = \#\mathbf{0}$$

The BDD in figure 7.10 represents the following type:

$$[\,\mathit{integer}() \mid \mathit{atom}()\,] \sqcap \neg\,[\,\mathit{number}() \mid \neg\mathit{integer}()\,]$$

As the BDD in figure 7.10 is unsatisfiable, the sub-typing relation holds.

The modified `restrict` and `if-then-else` algorithms use the sub-typing relation to remove redundant type tests from BDDs as they are constructed, ensuring that semantically equivalent types are represented by structurally identical BDDs. As $S \leqslant T$ holds when $S \sqcap \neg T = \neg\mathit{term}()$, and as $\neg\mathit{term}()$ is represented by a single #**0** leaf, the sub-typing relation can be determined by checking whether the result of $\texttt{Add-Type}(S \sqcap \neg T)$ is a pointer to #**0**.

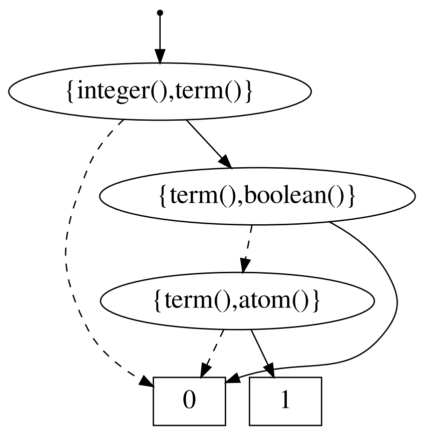


Figure 7.11: Type BDD for $[\,\mathit{integer}() \mid \mathit{atom}()\,] \sqcap \neg [\,\mathit{number}() \mid \mathit{boolean}()\,]$

7.3.1 Producing Counter-examples for Sub-Typing

The sub-typing relation $S \leqslant T$ holds when the BDD for $S \sqcap \neg T$ is unsatisfiable, i.e. when there is no path from the root node to the 1 leaf. When the BDD *is* satisfiable the sub-typing relation does not hold, and furthermore the path from the root note to the 1 leaf provides a counter-example to the sub-typing relation.

For example, the BDD in figure 7.11 represents the following type:

$$[\,\mathit{integer}() \mid \mathit{atom}()\,] \sqcap \neg [\,\mathit{number}() \mid \mathit{boolean}()\,]$$

The sub-typing relation does not hold in this case, and by following the path from the root to the 1 leaf we obtain the following type (by taking the intersection of the types found along the path):

$$\begin{aligned} &= [\,\mathit{integer}() \mid \mathit{term}()\,] \sqcap \neg [\,\mathit{term}() \mid \mathit{boolean}()\,] \sqcap [\,\mathit{term}() \mid \mathit{atom}()\,] \\ &= [\,\mathit{integer}() \mid \mathit{atom}()\,] \sqcap \neg [\,\mathit{term}() \mid \mathit{boolean}()\,] \end{aligned}$$

This type represents the set of values in S which are not also members of T: 2 element tuples whose first element is an $\mathit{integer}()$ and whose second element is an $\mathit{atom}()$ but *not* a $\mathit{boolean}()$.

These counter-examples can be used to give additional diagnostic information wherever the sub-typing relation is being used: if we expect the sub-typing relation to hold and it does not, we can provide counter-examples to the programmer to assist them with debugging.

Chapter 8

Hybrid Verification of Erlang Communications

Back in chapter 4 we presented a communicating fragment of Core Erlang called CoErl. The language was intended to serve as a formal model of Erlang's pattern matching and guard evaluation, and as an environment for reasoning about Erlang's asynchronous message passing behaviour. In chapter 5 CoErl served as the basis for a trace theory which modelled communications as a sequence of events, which eventually led to the development of a sub-typing system for CoErl in chapter 6.

Each of these ideas has been presented in an abstract setting, in isolation from the real Erlang implementation as seen in Erlang/OTP. Furthermore, we haven't used these ideas to address the central topic of this thesis: communication discrepancies in Erlang programs. This chapter addresses both of these points together in the form of a *hybrid analysis* of Erlang programs with the goal of *automatically* detecting communication discrepancies. The analysis is a combination of static analysis and runtime verification. Using static analysis we can easily detect some discrepancies based on violations of the sub-typing relation, while runtime verification allows us to intercept messages which will crash processes. The two techniques are complementary because while static analysis enables early detection of errors, runtime verification can be used where static analysis cannot easily decide whether or not communication is "safe".

The verification system will be written in Erlang as it offers first-class access to the Erlang compiler, virtual machine, standard library, and type handling system.

Overview We first present the basis of the hybrid analysis - a notion of *message compatibility* which we will use to reason about whether a sent message is compatible with a given `receive` expression (section 8.1).

Then we consider the two distinct aspects of the hybrid analysis in turn: static analysis (section 8.2) and runtime verification (section 8.3). With regards to static analysis, we look at how we can infer types for sent and received messages in a system at compile time. In addition, we use the sub-typing relation can be used to improve upon the Erlang compiler's existing mechanisms for detecting dead clauses and redundant type tests (sections 8.2.2 and 8.2.3).

We then turn our attention to runtime verification. First, we look at how communication discrepancies can occur at runtime in ways that cannot be detected at compile time. Specifically, we will examine how server processes written using the `gen_server` library behave when combined with advanced Erlang/OTP features such as code reloading (section 8.3.1). This is followed by a description of how runtime type checking can be performed to protect generic server processes from certain classes of runtime type errors (section 8.3.2).

An overview of an implementation follows, describing how each of the concepts from the static analysis and runtime verification sections can be implemented in Erlang, including the sub-typing relation which forms a core part of the analysis (section 8.4). We look at how message types can be inferred at compile time and how this information can be stored for access at runtime. Finally, we look at a lightweight implementation of `gen_server` which is then modified to check the types of incoming messages at runtime in order to protect server processes from communication discrepancies which could lead to crashes. This is accompanied by a brief report on modifying the real `gen_server` module so that it also performs runtime type checking.

8.1 Message Compatibility

In order to detect communication discrepancies in Erlang programs we must consider the relationship between messages sent to a process and the messages received by that process. For example, if a message is sent to a process which never receives it, that message will linger in the mailbox until the process exits. Likewise, if a process attempts to receive a message that was never sent to it then the process will wait – potentially forever.

These discrepancies are not always as clearly defined as something not being sent or something not being received because programming errors can lead to a subtle discrepancy where a message which *appears* to be receivable is in fact not, perhaps due to a typo or a transposition of elements in a tuple. For example, the function `start/0` in listing 10a on the next page spawns a server process and then becomes the client. The client sends the message `{get, From}` to the server, but the server is expecting a 3-element tuple of the form `{get, From, Ref}`. When we

```
-module(discrep1).
-export([start/0,server/1,client/1]).

server(State) ->
 receive
  {get, From, Ref} ->
   io:fwrite("Server responding~n"),
   From ! {resp, State, Ref},
   server(State)
 end.

client(Server) ->
 io:fwrite("Client sending request~n"),
 Server ! {get, self()},
 % missing element above
 receive
  {resp, St, _Ref} ->
   io:fwrite("Client got ~p~n", [St])
 end.

start() ->
 Server = spawn(?MODULE, server, [42]),
 client(Server).
```

(a) Client with wrong request format

```
-module(discrep2).
-export([start/0,server/1,client/1]).

server(State) ->
 receive
  {get, From, Ref} ->
   io:fwrite("Server responding~n"),
   From ! {resp, State, Ref},
   io:fwrite("Server sending 'done'~n"),
   From ! done
 end,
 io:fwrite("Server exiting~n").

client(Server) ->
 Ref = make_ref(),
 io:fwrite("Client sending request~n"),
 Server ! {get, self(), Ref},
 receive
  {resp, S, Ref} ->
   io:fwrite("Client received ~p~n", [S])
 end,
 timer:sleep(1000), % wait a little bit

 Key = message_queue_len,
 {_, N} = process_info(self(), Key),
 io:fwrite(
  "~p messages in mailbox ~n", [N]).

start() ->
 Server = spawn(?MODULE, server, [42]),
 client(Server).
```

(b) Server sends an unreceived message

Listing 10: Erlang programs with message compatibility ussues

run this program we see that the client never receives a reply from the server because of this initial communication discrepancy:

```
1> discrep1:start().
Client sending request
<hangs here>
```

The client will wait for a response forever. If the client's `receive` expression contained a finite timeout the program would eventually continue executing, but the discrepancy would still exist: the type of the request send by the client is *incompatible* with the type of the `receive` clause in the server.

In listing 10b on the preceding page we have a different problem where the server sends a `'done'` message to the client in addition to the response:

```
1> discrep2:start().
Client sending request
Server responding
Server sending 'done'
Client received 42
Server exiting
1 messages in mailbox
ok
2> process_info(self(), message_queue_len).
{message_queue_len,1}
```

The extra `'done'` message will linger in the mailbox until it either exits, or another piece of code happens to receive a message of the same type (which might interfere with some other communications).

Both of these discrepancies can be reasoned about using the sub-typing relation from chapter 6. In listing 10a the type of the sent message by the client is $\{\texttt{'req'},\mathtt{pid}()\}$ and the type of message accepted by the the `receive` clause in the server is $\{\texttt{'req'},\mathtt{term}(),\mathtt{term}()\}$ based on pattern type inference. Note that the sub-typing relation *does not* hold here:

$$\{\texttt{'req'},\mathtt{pid}()\} \nleqslant \{\texttt{'req'},\mathtt{term}(),\mathtt{term}()\}$$

In the case of listing 10a we see that the type of the request is $\{\texttt{'req'},\mathtt{pid}(),\mathtt{reference}()\}$ and that the type of the `receive` clause is the same as before. Here, the type of the sent message is a sub-type of the inferred `receive` type:

$$\{\texttt{'req'},\mathtt{pid}(),\mathtt{reference}()\} \leqslant \{\texttt{'req'},\mathtt{term}(),\mathtt{term}()\}$$

Therefore, we can approximate whether a message is compatible with a `receive` expression via the sub-typing relation: a message is compatible with a `receive` expression if the type of the message is a sub-type of the `receive` type. This will form the basis of the hybrid analysis: we will use the sub-typing relation to reason about whether sent messages are compatible with `receive` expressions both at compile time and runtime.

```
loop(N) ->
  receive
    {add, M} when is_number(M) ->
      loop(N+M);
    {sub, M} when is_number(M) ->
      loop(N-M);
    {get, From, Ref} ->
      From ! {res, N, Ref},
      loop(N);
    stop ->
      io:fwrite("stopping.~n"),
      ok
  end.
```

(a) Server code

```
client1(Server) ->
  Server ! {add, 10},
  Server ! {sub, 5},
  Server ! {get, self()}.

client2(Server) ->
  Server ! {add, 10},
  Server ! {sub, hello},
  Ref = make_ref(),
  Server ! {get, self(), Ref},
  receive
    {res, State, Ref} ->
      io:fwrite("Got ~p~n", [State])
  end.
```

(b) Client code

Listing 11: Counter server and client with communication discrepancies

8.2 Static Analysis

Certain checks can be performed automatically at compile time. Using the sub-typing system from chapter 6 we can detect certain kinds of communication discrepancies at compile time. These checks, however, are somehwat limited due to Erlang's "open world" communication model where any process can communicate with any other. For example, a data flow analysis of PIDs used to ascertain which messages are sent to which process will likely be unable to track PIDs obtained via a registered name lookup, retrieved from an ETS table, or if the lookup depends on data provided via runtime configuration. We can nonetheless infer types for sent and received messages using our sub-typing system.

The sub-typing system can also be used to detect other kinds of "code smell" entirely automatically. Namely, we can detect dead clauses in `case` and `receive` expressions, and we can improve upon the compiler's ability to detect redundant type tests in guard expressions across multiple clauses.

8.2.1 Message Compatibility

In chapter 3 on page 38 we considered the different kinds of communication discrepancies that can occur in Erlang programs. These discrepancies fell into two main categories: messages sent but never received, and messages received that were never sent. The code in listing 11 exhibits both of these types of communication discrepancy, and we can detect most of them with sub-typing alone.

To determine whether a discrepancy exists we consider each sent message with all of the `receive` expressions in the body of the recipient. For each message sent we expect to see a receive clause *capable* of matching that message. If there is no such clause we know that the message *will never* be received. Likewise, if we see a `receive` clause and never see a sent message which can match it, then we can say that clause is "unused".

In terms of types, we consider the type of each sent message with the type of each `receive` clause in the recipient process' code. For each sent message with type S we expect to find at least one `receive` clause with a type T such that $\mathsf{S} \leqslant \mathsf{T}$. On the other hand for each `receive` clause with type T we expect to find a sent message which has a type S such that $\mathsf{S} \leqslant \mathsf{T}$.

We follow with some examples of how we can detect the discrepancies in listing 11:

Messages sent but never received The `client1` function sends a 2-element tuple which will never be received by the server. That is, for the set of all receive clause types in the server:

$$\mathsf{Ts} = [\{\texttt{'add'}, \mathit{number}()\}, \{\texttt{'sub'}, \mathit{number}()\}, \{\texttt{'get'}, \mathit{term}(), \mathit{term}()\}, \texttt{'stop'}]$$

there is no clause which will accept the message:

$$\nexists \mathsf{T} \in \mathsf{Ts}. \{\texttt{'get'}, \mathit{term}()\} \leqslant \mathsf{T}$$

There is a similar situation in `client2` involving the `{sub, hello}` message:

$$\nexists \mathsf{T} \in \mathsf{Ts}. \{\texttt{'get'}, \texttt{'hello'}\} \leqslant \mathsf{T}$$

Messages received but never sent The `client1` function never sends a proper `get` or `stop` message to the server. For the set of all message types sent to the server:

$$\mathsf{Ss} = [\{\texttt{'add'}, 10\}, \{\texttt{'sub'}, 5\}, \{\texttt{'get'}, \mathit{term}()\}]$$

there is no sent message which could ever match either of the clauses:

$$\nexists \mathsf{S} \in \mathsf{Ss}. \mathsf{S} \leqslant \{\texttt{'get'}, \mathit{term}(), \mathit{term}()\}$$
$$\nexists \mathsf{S} \in \mathsf{Ss}. \mathsf{S} \leqslant \texttt{'stop'}.$$

Again, there is a similar discrepancy between `client2` and the server where the `sub` clause of the `receive` is never used:

$$\nexists \mathsf{S} \in \mathsf{Ss}. \mathsf{S} \leqslant \{\texttt{'sub'}\mathit{number}()\}$$

Whenever we detect one of these discrepancies it should be presented to the programmer. These discrepancies often occur in pairs, too: an unreceived message discrepancy is usually accompanied by an unsent message discrepancy. One thing we cannot do however is assign blame because we do not know which (if any) of the implementations is correct. We have no type annotation or behavioural contract to work from, so the best thing to do is to report the discrepancies to the programmer as *early* as possible so that they can choose the best course of action. The aim of the analysis is not to reject programs or force a specific programming style upon Erlang users, but rather to alert them to communication discrepancies in their programs which are likely to affect behaviour of cause memory leaks.

One important and unresolved question is how to determine which sent message corresponds with which `receive` expression, if any. A data flow analysis could be used to track PIDs which are returned from calls to `spawn` which would allow us to determine which process a message is sent to, but it would not allow us to determine exactly which `receive` expression will receive the message, if any. An alternative approach would be to only perform analysis on processes with registered names, i.e. locally or globally registered unique identifiers which map to at most one PID.

Regardless of the approach used, a data flow analysis combined with type inference must be used on the sender's code to determine the types of messages which can be sent, and a call graph of the receiving code must be constructed to enumerate all receive expressions. This also presents an interesting question concerning the separation of code into separate modules and the conceptual boundaries that often accompany them: as both the sender and receiver can call functions in other modules which perform their own communications, should we consider message compatibility across all of these modules, even if the programmer did not write them? With respect to correctness it would perhaps be best to analyse *all* sent and received messages to find as many discrepancies as possible. The risk is that we alert the programmer to discrepancies which they have no ability to correct, or would be unable to analyse modules for which no source code is available[1]. On the other hand, if we only consider discrepancies within the programmer's own code (i.e. within a single module) our analysis becomes more straightforward, especially as we are guaranteed to have source code available. The disadvantage is that some communication discrepancies may go undetected since we do not consider how code in other modules sends or receives messages which in the latter case may affect the execution of the current process.

This chapter uses the latter approach: only code within a module will be analysed at compile time. Fortunately, this works well for Erlang modules which follow the convention of abstracting away all implementation-specific functionality behind a clear API. For example, modules which implement the `gen_server` behaviour typically export functions for interacting with the server

[1]Erlang modules are compiled independently and therefore it is possible to use modules which are only available in bytecode form.

instead of requiring that clients make calls to the `gen_server` module directly. In addition, when behaviours like `gen_server` are used by the programmer the analysis becomes even more straightforward because we know in advance which callback function corresponds to each request.

8.2.2 Dead Clause Detection

In section 2.1.1 we noted that the Erlang compiler can already detect some kinds of dead clauses: those which appear after a "wildcard" clause. We can improve upon this check using our sub-typing relation by checking whether the intersection of a type's clause and the *negation* of the types of all prior clauses is empty. For example, with the following `receive` expression we can use the sub-typing relation to detect that the third clause is dead:

```
receive
  X when is_integer(X) -> e1;
  X when is_float(X) -> e2;
  X when is_number(X) -> e3
end.
```

The inferred type of the first clause is $\mathtt{integer}()$ based on the `is_integer()` type test. Likewise, the inferred type of the second clause is $\mathtt{float}()$. However, we can also say that the type of the second clause is the inferred type of the second clause *and not* any of the types of the previous clauses, i.e. $\mathtt{float} \sqcap \neg\mathtt{integer}()$. We can do this because in order for a value to match the second clause it must not have matched the first clause.

When we repeat this process for the third clause we infer the type $\mathtt{number}()$ and intersect it with the negation of $\mathtt{float}()$ and $\mathtt{integer}()$:

$$\begin{aligned} & \mathtt{number}() \sqcap \neg(\mathtt{float}() \sqcup \mathtt{integer}()) \\ = {} & \mathtt{number}() \sqcap \neg\mathtt{number}() \\ = {} & \emptyset \end{aligned}$$

By using this technique to analyse Erlang code we can detect dead clauses before we ever get to the stage of checking for communication discrepancies, allowing us to provide the programmer with more information at an earlier time.

8.2.3 Pattern and Guard Refinement

```
-module(redundant).
-export([f/1]).

f(X) when is_atom(X) ->
  branch_one;
f(X) when not is_boolean(X) ->
  branch_two.
```

(a) Erlang source code

```
{function, f, 1, 2}.
  {label,1}.
    {line,[{location,"redundant.erl",4}]}.
    {func_info,{atom,redundant},{atom,f},1}.
  {label,2}.
    {test,is_atom,{f,3},[{x,0}]}.
    {move,{atom,branch_one},{x,0}}.
    return.
  {label,3}.
    {test,is_boolean,{f,4},[{x,0}]}.
    {jump,{f,1}}.
  {label,4}.
    {move,{atom,branch_two},{x,0}}.
    return.
```

(b) Compiled to human-readable BEAM bytecode

Listing 12: Erlang function with a redundant type test in a guard

Another optimisation we can perform with our type system is the elimination of redundant patterns and guards from clauses. As an example, consider the code in listing 12a: the first clause matches all atoms and the second clause matches all values which are *not* booleans. The second clause will *always* match because all booleans will be handled by the first clause as $\mathtt{boolean}() \leqslant \mathtt{atom}()$. Therefore, the guard in the second clause is redundant because as `not is_boolean(X)` will always return true.

Despite this, the Erlang compiler does not detect the redundant test and it is present in the BEAM bytecode generated by it. This bytecode (shown in listing 12b) is an imperative assembly-like version of `f`. Label 2 represents the first clause of the function: if the test `is_atom` fails, then we jump to label 3 otherwise we return `'branch_one'`. Label 3 represents the second clause where we check whether `X` is a boolean: if it is not a boolean then we return `'branch_two'`, otherwise we raise an exception stating that no function clause has matched.

If we look at the types of the clauses we can see that the second clause makes *no change* to the existing type constraints. The first clause has type $\mathtt{atom}()$ and the second clause has type $\neg\mathtt{boolean}()$. Therefore, *after* the first clause we know that the argument X cannot be an atom because it would've matched the first clause:

$$\neg\mathtt{atom}()$$

Then when we add the type for the second clause we notice that the type is semantically equivalent to what it was already:

$$\begin{aligned} & \neg\mathsf{boolean}() \sqcap \neg\mathsf{atom}() \\ = & \neg\mathsf{atom}() && \text{as } \mathsf{boolean} \leqslant \mathsf{atom}() \end{aligned}$$

As the new clause doesn't change the type we *already have*, we know that its patterns and guards can be removed and replaced with wildcards without affecting which values will match it. In this example we can remove the `not is_boolean()` guard from the second clause without affecting the behaviour of the function.

8.3 Runtime Verification

To address the limitations of static analysis we will complement it with runtime verification where we will check the types of incoming messages at runtime to ensure they do not constitute a communication discrepancy. This combines compile-time type inference with runtime instrumentation to intercept messages before they crash a process. We will examine how communication discrepancies can exist in the gen_server library and how advanced features such as live code reloading can make static analysis an almost futile task. In addition, we look at how we can perform type checking at runtime rather than compile time.

8.3.1 Communication Discrepancies in gen_server

Servers implemented using the gen_server library are particularly susceptible to communication discrepancies because their callback functions typically cannot handle unexpected requests, instead raising a function clause exception.

In chapter 3 on page 38 we saw that server processes such as the one implemented in listing 7 on page 25 will crash when they are sent a message of the wrong format or type:

```
3> {ok, ServerGen} = my_counter_gen:start(0).
{ok,<0.81.0>}
4> my_counter_gen:add(ServerGen, hello).
ok
=ERROR REPORT====
** Generic server <0.81.0> terminating
** Last message in was {'$gen_cast',{add,hello}}
** When Server state == 0
** Reason for termination ==
** {badarith,[{erlang,'+',[0,hello],[]},
              {my_counter_gen,handle_cast,2,
                              [{file,"my_counter_gen.erl"},{line,23}]}
              [...]]}
=CRASH REPORT====
  crasher:
    initial call: my_counter_gen:init/1
    pid: <0.82.0>
    registered_name: []
    exception error: an error occurred when evaluating an arithmetic expression
      in operator  +/2
         called as 0 + hello
      in call from my_counter_gen:handle_cast/2 (my_counter_gen.erl, line 23)
      in call from gen_server:try_dispatch/4 (gen_server.erl, line 637)
      [...]
```

These crashes occur due to the violation of an *implicit* behavioural contract between the server and the client, i.e. that the client will send well-formed requests and that the server will send a reply when one is expected. When the server violates this implicit contract the client is likely to block when waiting for a response, but when the client violates the contract the server is likely to crash, taking all of its state with it.

The nature of these crashes poses an interesting question because although the client violated this implicit contract, the server crashed. Should the server crash due to the client violating the contract? As before when we were performing a static analysis, we do not know whether the client implementation or the server implementation is correct, if any. Once again we will therefore focus on alerting the programmer to these discrepancies, except this time we will additionally prevent the server from crashing.

Live Code Reloading

We will also be able to detect communication discrepancies which occur due to a live code reload, where an Erlang module is replaced with a new version at runtime *while the application is executing*. Discrepancies can occur during a live code reload even if each version of the module has no communication discrepancies, but where discrepancies in the format and type of messages exists *between* the two versions.

```erlang
-module(my_counter_gen).
-behaviour(gen_server).
-vsn(1).

-export([start/1]).
-export([init/1,
         handle_call/3,
         handle_cast/2]).

start(N) ->
  gen_server:start(?MODULE, [N], []).

init([N]) ->
  {ok, N}.

handle_cast({add, N}, State) ->
  {noreply, State+N};
handle_cast({sub, N}, State) ->
  {noreply, State-N}.

handle_call(get, _From, State) ->
  {reply, State, State}.
```

(a) Version 1 of my_counter_gen

```erlang
-module(my_counter_gen).
-behaviour(gen_server).
-vsn(2).

-export([start/1]).
-export([init/1,
         handle_call/3,
         handle_cast/2]).

start(N) ->
  gen_server:start(?MODULE, [N], []).

init([N]) ->
  {ok, N}.

handle_cast({op, Op, N}, State) ->
  State1 = case Op of
             add -> State+N;
             sub -> State-N
           end,
  {noreply, State1}.

handle_call(get, _From, State) ->
  {reply, State, State}.
```

(b) Version 2 of my_counter_gen

As an example of how code reloading can affect communications in a system we will consider the my_counter module from listing 7 on page 25 (which has been duplicated in listing 13a for ease of reference). We interact with the server as before by calling the cast and call functions in the gen_server module. In addition, we now check the version of the module loaded in the virtual machine, which is taken from the vsn attribute:

```
1> {ok, Server} = my_counter_gen:start(0).
{ok,<0.80.0>}
2> gen_server:cast(Server, {add, 20}).
ok
3> gen_server:cast(Server, {sub, 5}).
ok
4> gen_server:call(Server, get).
15
5> beam_lib:version(my_counter_gen).
{ok,{my_counter_gen,[1]}}
```

In listing 13b we have a different version of our counter where we have decided that it is excessive to have two different request formats for adding and subtracting numbers. Instead we use a single 3-element `op` request format whose second element is the name of the operation, and the third is the amount to add or subtract by. Again, we can interact with this server using `call` and `cast`:

```
% start a new erlang shell
1> {ok, Server} = my_counter_gen:start(0).
{ok,<0.80.0>}
2> gen_server:cast(Server, {op, add, 20}).
ok
3> gen_server:cast(Server, {op, sub, 5}).
ok
4> gen_server:call(Server, get).
15
5> beam_lib:version(my_counter_gen).
{ok,{my_counter_gen,[2]}}
```

Each of these two versions of the counter work as expected when they are sent messages of the correct format, but a serious communication discrepancy can occur if we load the new version of the module at runtime when a counter server is already running.

First, we start a counter using version 1 of the module (listing 13a):

```
1> code:load_file(my_counter_gen).
{module,my_counter_gen}
3> {ok, Server} = my_counter_gen:start(0).
{ok,<0.82.0>}
4> gen_server:cast(Server, {add, 20}).
ok
5> beam_lib:version(my_counter_gen).
{ok,{my_counter_gen,[1]}}
```

Then we load the second version of the module (listing 13b) but we continue to use the request format from the first version, simulating a race condition:

```
6> % place new version of module in code path
7> code:load_file(my_counter_gen).
```

```
{module,my_counter_gen}
8> beam_lib:version(my_counter_gen).
{ok,{my_counter_gen,[2]}}
9> gen_server:cast(Server, {sub, 5}).
ok
10> =ERROR REPORT==== 16-Jan-2020::16:38:11.351345 ===
** Generic server <0.82.0> terminating
** Last message in was {'$gen_cast',{sub,5}}
** When Server state == 20
** Reason for termination ==
** {function_clause,[{my_counter_gen,handle_cast,
                                     [{sub,5},20],
                                     [{file,"my_counter_gen.erl"},{line,16}]},
```

When we loaded the second version of the `my_counter_gen` module the machinery of the `gen_server` library automatically started using the new version of the code. This meant that the server automatically transitioned from the code in listing 13a to the code in listing 13b with the request format changing in the process. Then, we sent a message to the server using the old request format, crashing it.

These race conditions can occur during live code reloads for a variety of reasons:

- the client code might be in a separate module, meaning that malformed requests will be sent until the client code is also reloaded; or
- the client code and server code might be in the same module, but a request message might be "in flight" while the code is being reloaded; or
- the client code could be executing *during* the code reload, meaning that the old request format will be used even if the new code is loaded.

There are several ways to avoid this race condition such as manually ensuring that the second version of a module is compatible with the first version, manually flushing the server's mailbox during a code reload (and therefore causing clients waiting on responses to time out), or perhaps simply allow the server to crash.

8.3.2 Type Checking

Runtime verification offers a method of automatically verifying whether a message is compatible with a server process at runtime, allowing us to bypass issues which only exist at compile time such as determining which version of code will be running, or determining exactly which processes will be communicating with the server and how. By intstrumenting the `gen_server` module we can perform type checking at runtime before the messages are passed to callback functions, allowing us to reject messages which would definitely crash the server. These generic behaviours are an ideal candidate for instrumentation because of the separation of concern between callbacks, the exposed API, and internal functionality. For example, if a bug is fixed in the `gen_server` module and the fixed version is loaded into the BEAM, all modules which implement the behaviour will benefit from the bugfix automatically without being recompiled.

Assuming that the type of an incoming message is S and the inferred type of messages accepted by the corresponding callback function is T, then the message will be passed to the callback function if and only if $S \leqslant T$. The type S can be determined at runtime by inspecting the message directly: at runtime the message is a concrete Erlang term, not an expression. To infer the types of messages accepted by the callback function we use exactly the techniques described in section 6.4: infer the types of patterns, guards, clauses, and then sequences thereof.

Unfortunately, it is difficult to inspect the source code of a function at runtime and in many circumstances it may not even be available. To address this we can perform fully automatic compile-time type inference on the callback functions to determine the types of messages they will accept, and then we can automatically embed this metadata in the module so that it is available at runtime.

8.4 Implementation

The remainder of this chapter gives an overview of the implementation of these ideas for hybrid verification. First, we must implement the sub-typing algorithm chapter 7 by transliterating the algorithms into Erlang, taking advantage of graph libraries available in Erlang/OTP. Then we look at how we can statically infer message types from Core Erlang source code, discussing some quirks of the Core Erlang specification and the Erlang/OTP compiler which complicate the analysis. These inferred types are then made available at runtime by automatically generating and injecting a function definition into the module under analysis. This function can be called at runtime to retrieve the statically inferred message types for a `gen_server` callback function.

Bringing all of this together, we then look at a lightweight model of the `gen_server` module and instrument it to perform runtime type checking using a combination of sub-typing and the inferred callback types. The result is a fully automatic system which infers types of messages accepted by callbacks at compile time, automatically injects this information into modules during compilation, and then performs runtime type checking using the sub-typing algorithm to prevent crashes caused by message incompatibility.

8.4.1 Sub-Typing

In chapter 7 we dispensed with a rewrite-based approach and instead used BDDs to represent types. An important property of these BDDs is that they are canonical: any two semantically equivalent types are represented by structurally equivalent BDDs.

The algorithms presented in that chapter were also *stateful*: they mutated data structures without returning them. It is easier to implement the `Restrict` and `If-Then-Else` algorithms this way because it avoids having to propagate the state of the MRDAG and memoisation table between every function call. Unfortunately Erlang does not allow us to mutate variables nor maintain any global variables, which presents a problem for our implementation of these algorithms.

The solution is *Erlang Term Storage* (ETS), a tabular datastore built into the Erlang runtime system (Erlang/OTP Team 2019b, Tables and Databases). Using ETS we can represent a mutable global state easily, and we can take advantage of all the optimisations and algorithms used for efficient insertion and lookup.

ETS tables allow us to represent the memoisation tables used by `If-Then-Else` easily, but not the MRDAG which actually contains the BDD. Fortunately, the `digraph` module in the standard library does exactly what we need: it allows us to insert, update, and delete nodes and edges in a directed acyclic graph, returning *identifiers* of nodes rather than the nodes themselves . This library even uses ETS tables to represent the graphs it creates, so we get global mutable state for free. The rest of the sub-typing implementation is a transliteration of the positive type intersection function, the `If-Then-Else` algorithm, and the `Restrict` algorithm.

The result is a function called `is_subtype/2` which takes two types as inputs and returns a boolean value:

```
1> types:is_subtype({cons,integer,atom},{cons,number,boolean}).
false
2> types:is_subtype({cons,integer,atom}.{cons,number,{neg,integer}).
true
```

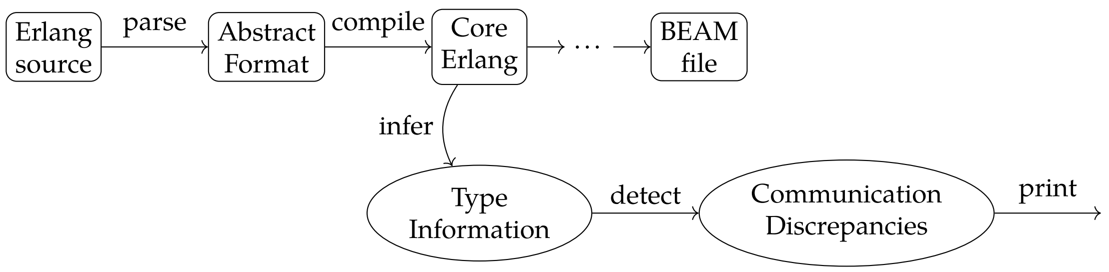


Figure 8.1: Core Erlang's position in the Erlang compiler

8.4.2 Type Inference

To determine message compatibility automatically we will need to infer the types of messages and `receive` expressions. In section 6.4 we presented algorithms for inferring the types of CoErl patterns, guards, and clauses. Here we will operate on real Erlang code, so we will need to write an Erlang implementation of the type inference algorithms that operates on Erlang abstract syntax trees.

To make this process easier we will work on the *Core Erlang* representation of a module because it removes a large amount of Erlang's syntactic sugar, normalises function definitions, and prohibits bound or repeated variable names in patterns.

The Erlang compiler translates the functional syntax of Erlang code into imperative BEAM bytecode via a number of intermediate formats. Core Erlang is one of these intermediate formats, originally created for the purposes of making static analysis of Erlang code easier as part of the *HiPE* project (Kostis Sagonas et al. 1998). The current version of the Erlang compiler translates Erlang source code into Core Erlang after expanding macro definitions and before converting to other low-level formats. The diagram in figure 8.1 shows how we will be using Core Erlang for our static analysis: we will compile Erlang source code to Core Erlang, infer types of sent and received messages, and use it to detect communication discrepancies.

We normally compile Erlang source code to BEAM bytecode without outputting any intermediate representations, but command line flags allow us to stop compilation once we reach Core Erlang:

```
$ ls
my_module.erl
$ erlc +to_core my_module.erl
$ ls
my_module.core my_module.erl
```

This gives us a *textual* version of Core Erlang code in the `my_module.core` file.

For programmatic access though, we can do better: the Erlang standard library includes the `compile` module which exposes the required APIs (Erlang/OTP Team 2019a). For example, we can replicate the behaviour of the `erlc` command:

```
$ ls
my_module.erl
$ erl
Erlang/OTP 22 [erts-10.6] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]

Eshell V10.6  (abort with ^G)
1> compile:file(my_module, [to_core]).
{ok,my_module}
2> q().
ok
$ ls
my_module.core  my_module.erl
```

We are however still creating a textual representation of the Core Erlang syntax tree in the filesystem. To obtain a representation of the Core Erlang AST *in Erlang* we use the `binary` option. Furthermore, `debug_info` instructs the compiler to maintain debug information such as source file locations and line numbers as source code annotations:

```
1> compile:file(my_module, [to_core,binary,debug_info]).
{ok,my_module,
    {c_module,[],
        {c_literal,[],my_module},
        [{c_var,[],{f,0}},
         {c_var,[],{module_info,0}},
         {c_var,[],{module_info,1}}],
        [{{c_literal,[1],file},
          {c_literal,[1],[{"my_module.erl",1}]}},
        ...
```

If the compilation to core Erlang succeeds then a tuple is returned where the second element is the name of the module we just compiled, and the third element is the Core Erlang AST for that module. Each node is a tagged tuple, where the first element of the tuple identifies the type of

```
pattern_type(P,Gamma) ->
  case cerl:type(P) of
    'var' ->
      case Gamma(cerl:var_name(P)) of
        {ok, T} -> T;
        fail    -> term
      end;
    'literal' ->
      Value = cerl:concrete(P),
      type_of(Value);
    'tuple' ->
      Types = [ pattern_type(E,Gamma) || E <- cerl:tuple_es(P) ],
      {tuple, Types};
    'cons' ->
      HeadT = pattern_type(cerl:cons_hd(P), Gamma),
      TailT = pattern_type(cerl:cons_tl(P), Gamma),
      {cons, HeadT, TailT};
    'alias' ->
      PatT = pattern_type(cerl:alias_pat(P),Gamma),
      case Gamma(cerl:alias_var(P)) of
        {ok, VarT} -> intersection(PatT, VarT);
        fail -> PatT
      end;
    _Ty ->
      error({unsupported_pattern_type, P})
    end.
```

Listing 14: Type inference function for Core Erlang patterns

node (e.g. `'c_literal'` for "Core Erlang literal"). Instead of manually deconstructing each of these tuples when performing type inference we will use the `cerl` module from the compiler library (Carlsson 2019):

```
1> {ok, _, Mod} = compile:file(my_module, [to_core,binary,debug_info]).
{ok,my_module,...}
2> Defs = cerl:module_defs(Mod)
[{{c_var,[],{f,0}},
  {c_fun,[4,{file,"my_module.erl"}],
         [],
         {c_literal,[5,{file,"my_module.erl"}],ok}}},
  ...
```

Patterns

```
type_of(X) when is_boolean(X)   -> {singleton, X, boolean};
type_of(X) when is_atom(X)      -> {singleton, X, atom};
type_of(X) when is_integer(X)   -> {singleton, X, integer};
type_of(X) when is_float(X)     -> {singleton, X, float};
type_of(X) when is_pid(X)       -> {singleton, X, pid};
type_of(X) when is_reference(X) -> {singleton, X, reference};
type_of(X) when X == []         -> nil;
type_of(X) when is_list(X)      -> {cons, type_of(hd(X)), type_of(tl(X))};
type_of(X) when is_tuple(X)     -> {tuple, [type_of(E) || E <- tuple_to_list(X)]}.
```

Listing 15: Function to determine the type of Erlang terms: `type_of`

The `pattern_type` function in listing 14 on the previous page is a transliteration of the $\mathcal{P}\,[\![\mathrm{p}]\!]_{\Gamma}$ function from figure 6.9 on page 110. The function takes a Core Erlang AST `P` and a typing environment `Gamma` and returns an Erlang representation of its inferred type.

We switch on `cerl:type(P)`, which returns an atom describing the type of node `P`. If the node is a variable we extract its name from the node and check whether it exists in the environment by calling it (`Gamma` is a function). If the variable is in the environment we return its type, otherwise we return `term`, representing the top type. When we encounter a literal (i.e. a concrete atom or integer) we retrieve its value from the node and return a singleton type as determined by the `type_of` function in listing 15. Tuples are represented as lists of AST nodes, so we infer the type of each element separately, bring them together into a list of types via a list comprehension, and return a tuple type. For cons cells we infer the types of the heads and tails separately, then create a cons type. Aliases of the form $\mathrm{p} = v$ are the only interesting case: we infer the type of the pattern p and intersect it with the type of v from the typing environment if it exists in the environment. Note that there is no case for empty lists here because nil is a *concrete value* and is therefore represented by `literal` nodes, which are handled in the second case.

Guards

Core Erlang guards are significantly more complicated to analyse compared to their relatively clean theoretical counterparts in section 6.4. This is because the Erlang compiler performs several syntactic transformations to Erlang guards so they can be represented in Core Erlang:

- The boolean operators `andalso` and `orelse` are transformed into semantically equivalent case expressions because these operators do not exist in Core Erlang.
- Expressions are wrapped in `try/catch` blocks when the compiler cannot guarantee that an expression will not raise an exception.

```
guard_type(G) ->
  case cerl:type(G) of
    'literal' ->
      case cerl:concrete(G) of
        'true'  -> {[env_empty()], [env_all(negate(term))]};
        'false' -> {[env_all(negate(term))], [env_empty()]};
        Lit -> error({unsupported_guard_literal, Lit})
      end;
    'call' -> guard_call_type(G);
    'case' -> guard_case_type(G);
    'try' -> guard_type(cerl:try_arg(G));
    'let' ->
      case let_is_boolean_coercion(G) of
        true -> guard_type(cerl:let_arg(G));
        false -> error({unsupported_let, G})
      end;
    _ -> error({unsupported_guard, G})
  end.
```

Listing 16: Type inference function for Core Erlang guards

- Expressions are coerced into boolean values when the compiler cannot guarantee that an expression always evaluates to a boolean value.

These transformations are necessary to satisfy the Core Erlang specification: guard expressions must not raise exceptions, and they must always return a boolean value.

The `guard_type` function in listing 16 performs the type inference for guards, returning the same data structure as $\mathcal{G}\,[\![G]\!]$ from section 6.4: a list of typing environments for which the guard returns true, and another list of environments for which it returns false. In the first case we handle the guards `'true'` and `'false'` identically to $\mathcal{G}\,[\![g]\!]$. We next handle function calls, which we expect to be one of the following:

- a type test BIF of the form `is_T(X)` for which we infer the type `T` for `X`; or
- a call to the non-short-circuiting boolean functions `and` and `not`, which we handle similarly to $\mathcal{G}\,[\![g]\!]$; or
- a call to the `=:=` function which either checks for equality between a variable and a literal (e.g. `X =:= 2`) or performs a boolean coercion

The next clause handles case expressions of the form:

```
case Exp of
  true -> TrueExp;
```

```
guard_case_type(G) ->
  case case_is_if_then_else(G) of
    {true, Test, True, False} ->
      {TestT, TestF} = guard_type(Test),
      {TrueT, TrueF} = guard_type(True),
      {FalseT, FalseF} = guard_type(False),
      {conj_env_list(TestT,TrueT) ++ conj_env_list(TestF,FalseT),
       conj_env_list(TestT,TrueF) ++ conj_env_list(TestF,FalseF)};
    false -> error({unsupported_case_expression, G})
  end.
```

Listing 17: Case expression type inference function for Core Erlang guards

```
  false -> FalseExp
end
```

which are treated similarly to if-then-else expressions from CoErl. Next, we handle `try/catch` blocks, and finally we attempt to remove any `let` bindings injected by the compiler.

For more insight into how these auxiliary functions behave, we consider boolean operator elision, exception handling, and boolean coercion in separate detail.

Boolean Operator Elision As Core Erlang does not have the short-circuiting `andalso` and `orelse` operators, nor an if-then-else expression like CoErl, the Erlang compiler uses case expressions to mimic their behaviour. For example, the Erlang expression `A andalso B` is equivalent to the following case expression:

```
case A of
  true -> B;
  false -> false
end
```

and `A orelse B` is equivalent to this case expression:

```
case A of
  true -> true;
  false -> B
end
```

The `guard_case_type` function in listing 17 is responsible for inferring the types of these expressions. First, we check whether the `case` AST node looks like an if-then-else: does it have a `true` clause and a `false` clause and nothing else? If so, we perform the same conjunction operation as the $\mathcal{G}[\![g]\!]$ function: we infer the type of the test, true branch, and false branch, then merge the environments together. Here, the `conj_env_list` function is equivalent to the $\wedge_{[\Gamma]}$ operator.

Exception Handling Some of the BIFs whitelisted for use in guard expressions can cause exceptions, such as the `hd` function:

```
1> hd(2).
** exception error: bad argument
     in function  hd/1
        called as hd(2)
```

As the Core Erlang specification forbids guard expressions to raise exceptions the Erlang compiler must somehow prevent this from occurring. Therefore, an expression like:

```
hd(X)
```

is converted to the following Core Erlang expression if used in a guard:

```
try
  ( let <_4> = ( call 'erlang':'hd' (X) ) in
  ( call 'erlang':'=:=' (_4,'true') )
  |- ['compiler_generated'] )
of <Try> -> Try
catch <T,R> -> 'false'
```

Furthermore, as Erlang guards which raise exceptions are considered failed (i.e. equivalent to returning `false`), wrapping the entire guard expression in a `try/catch` block does not change its behaviour.

Boolean Coercion The most significant transformation applied to guard expressions is boolean coercion. When the Erlang compiler cannot guarantee that an expression will return a boolean value it will wrap that expression with an equality check on the value `true`. For example, if we are not sure that the following function call returns a boolean:

```
my_mod:my_fun()
```

then we can wrap it with an equality check:

```
my_mod:my_fun() =:= true
```

This guarantees that a boolean value is *always* returned.

To make matters slightly more complicated, the compiler also inserts `let` expressions where there previously were none. For example, the Erlang expression:

```
erlang:is_integer(X)
```

may be abstracted into the following Core Erlang code:

```
( let <_4> = ( call 'erlang':'is_integer' (X) ) in
( call 'erlang':'=:=' (_4,'true') )
|- ['compiler_generated'] )
```

Fortunately, the compiler adds an annotation to the AST node to tell us that it has been generated, rather than being present in the original source code.

To handle these guards we remove the variable binding by substitution, which is safe to do because guard expressions do not have side effects

In all cases, the output of the `guard_type` function is a 2 element tuple of typing environments, where the first element is the environments in which the guard evaluates to `'true'`, and the second is the environments where it evaluates to `'false'`.

Assuming that `Node` is a Core Erlang AST representing the following Erlang expression:

```
is_integer(B) orelse is_boolean(B)
```

then the `guard_type` function produces the following output:

```
1> {Xs,Ys} = guard_type(Node).
{...,...}
2> [ Gamma('B') || Gamma <- Xs ].
[{ok,integer},{ok,{intersection,{neg,integer},boolean}}]
3> [ Gamma('B') || Gamma <- Ys ].
[{ok,{intersection,integer,{neg,term}}},
 {ok,{intersection,{neg,integer},{neg,boolean}}}]
```

The list `Xs` represents the successful evaluations of the guard: when `'B'` is an integer, or when it is not an integer but it is an atom. The list `Ys` represents the unsuccessful evaluations: when `'B'` is both an integer and not a term (i.e. never), or when it is both not an integer and not an atom.

```
clause_type(C) ->
  G = cerl:clause_guard(C),
  {Gammas,_} = guard_type(G),
  [Pat] = cerl:clause_pats(C), % only has one pattern
  Types = [ pattern_type(P, Gamma) || Gamma <- Gammas ],
  union_list(Types).
```

Figure 8.2: Type inference function for Core Erlang clauses

```
clauses_types(Cs) ->
  [ clause_type(C) || C <- Cs ].

receive_type(Rec) ->
  Types = clauses_types(cerl:receive_clauses(Rec)),
  lists:foldl(fun union/2, {neg,term}, Types).

receive_clauses_types(Rec) ->
  Types = clauses_types(cerl:receive_clauses(Rec)),
  {Res, _Acc} = lists:mapfoldl(fun(Type,Acc) ->
      Ty = intersection(Type,negate(Acc)),
      {Ty, union(Ty,Acc)}
    end, {neg,term}, Types),
  Res.
```

Figure 8.3: Type inference functions for Core Erlang receive expressions

Clauses

To infer the type of terms a clause accepts we again follow the process of transliterating from the mathematical definition, in this case $\mathcal{C}\,[\![\langle p\rangle\ \texttt{when}\ g \to e]\!]$ from figure 6.12 on page 115. The function `clause_type` in figure 8.2 performs this task: it infers the typing environments from the guard, uses the left-hand list of environments as inputs to the type inference function, and then takes a union of the resulting list.

Clause Sequences & Receive Expressions

The final task is to infer the type of messages accepted by a `receive` expression. From chapter 6 we know that this type is equal to the union of all of the expression's inferred clause types. This approach is covered by `receive_type` in figure 8.3 which folds over the list of inferred clause types, using the type $\neg\texttt{term}()$ as the initial accumulator (which is the identity element for union: $(\neg\texttt{term}()) \sqcup \mathsf{T} = \mathsf{T}$).

The more precise approach is to infer the type of each clause and then intersect the type of each clause with the union of all prior clause types. Back in chapter 6 we showed that this technique can be used to determine *which* clause a message will match and it also allows us to detect dead clauses at compile time. This is implemented by `receive_clauses_types` (also in figure 8.3) which returns a list consisting of each clause type successively intersected with the union of all previous types.

As an example of this type inference implementation, the following Erlang `receive` expression:

```
receive
  A when is_atom(A) -> a;
  B when is_integer(B) orelse is_boolean(B) -> b;
  {C,D} when is_atom(D) -> c
end
```

is inferred by `receive_type` to have this type:

```
{union,{union,{tuple,[term,atom]},{neg,term}},
       {union,{union,{intersection,{neg,integer},boolean},
                    {union,integer,{neg,term}}},
             {union,{union,atom,{neg,term}},{neg,term}}}}
```

which, using the notation from chapter 6, is equivalent to:

$$\begin{aligned} & \{\mathtt{term}(), \mathtt{atom}()\} \sqcup \emptyset \sqcup (\neg \mathtt{integer}() \sqcap \mathtt{boolean}()) \sqcup \mathtt{integer}() \sqcup \emptyset \sqcup \mathtt{atom}() \sqcup \emptyset \sqcup \emptyset \\ = \ & \{\mathtt{term}(), \mathtt{atom}()\} \sqcup \mathtt{boolean}() \sqcup \mathtt{integer}() \sqcup \mathtt{atom}() \\ = \ & \{\mathtt{term}(), \mathtt{atom}()\} \sqcup \mathtt{integer}() \sqcup \mathtt{atom}() \end{aligned}$$

Sent Messages

The type of a sent message can be approximated based on its structure by using the pattern type inference function `pattern_type` with an empty typing environment. In addition, we infer the type of any function call to be $\mathtt{term}()$ as our type inference system does not aim to type the functional part of Erlang. For example, we approximate that the send in the following function has type $\{\texttt{'add'}, \mathtt{term}()\}$:

```
add(Server, N) ->
  Server ! {add, N}.
```

We assume that every variable has type $\mathtt{term}()$ because we are not performing any data flow analysis, i.e. we know nothing about the type of `N` based on its usage or existing type information.

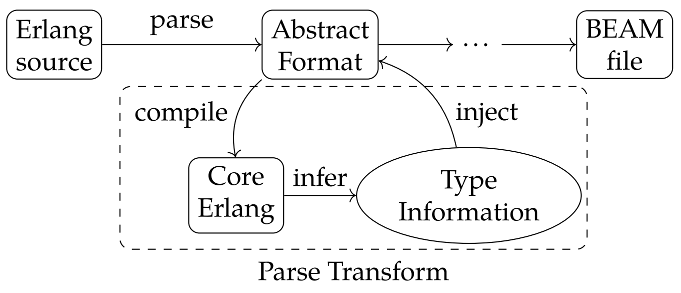


Figure 8.4: Parse Transform in Erlang compiler passes

```
Data: Metadata mapping
Result: type_info/1 function definition and export attribute in abstract format
function ⟵ new function node;
function.name ⟵ type_info;
function.arity ⟵ 1;
for (name, types) ∈ metadata map do
    clause ⟵ new function clause node;
    clause.arg[0] ⟵ function name/arity tuple to abstract format;
    clause.body ⟵ types converted to abstract format;
    append clause to function.clauses;
end
attr ⟵ new module attribute;
attr.name ⟵ export;
attr.value ⟵ [{type_info,1}];
return attr and function;
```

Algorithm 8: Generation process for `type_info/1` function

In this example we know nothing at all about `N`: any value could be passed to the function, so it is reasonable to assume that it could be anything. This ultimately leads to an overapproximation of the types of sent messages because data flow analysis might yield type information which allows us to refine the types of variables. Despite this it still allows us to detect communication discrepancies at compile time using the sub-typing relation.

8.4.3 Metadata Injection

With type information for each callback function in hand via compile-time type inference, we then inject it into the Erlang module as it is being compiled. To do this we will use a *parse transform* which can be used to perform arbitrary transformations to an Erlang AST as it passes through the compiler. The diagram in figure 8.4 shows where this transform occurs: after macro expansion, but before any other passes. The module is compiled to Core Erlang so we can perform our type inference, then we generate Erlang AST nodes for a new function called `type_info` which contains type information for every callback function. Algorithm 8 gives an overview of how we generate these new AST nodes: create a function AST node, create a new function clause for each callback which matches against the name of the callback, and generate a new export attribute.

Now we need to merge these new abstract nodes back into the original syntax tree (which is a list at the top level). The Erlang compiler uses the following approach when it injects the `module_info` functions:

```
add_predefined_functions(Forms) ->
    Forms ++ predefined_functions(Forms).
```

but we cannot do this because the above happens *after* internal compiler linter checks. If we attempt to naïvely append the new AST nodes we have generated then we get the following error from the compiler:

```
attribute export after function definitions
```

Instead we will write a comparison function `form_leq/2` which orders attribute nodes before function nodes. This allows us to merge the new function definition and export attribute into the module without falling foul of the linter:

```
merge_forms(Forms1, Forms2) ->
    lists:merge(fun form_leq/2, Forms1, Forms2).
```

This process automatically injects a function called `type_info` into modules which implement generic OTP behaviours which returns type information about that function when it is called. Using this transformation on version 1 of our counter module from listing 13a gives us the following function:

```
1> my_counter_gen:type_info({handle_call,3}).
[{literal,get,atom}]
2> my_counter_gen:type_info({handle_cast,2}).
[{tuple,[{literal,add,atom},term]},{tuple,[{literal,sub,atom},term]}]
```

8.4.4 Lightweight Model of gen_server

The real `gen_server` module is elaborate: it contains hundreds of lines of Erlang code which deal with error logging, timeout handling, scheduling, debugging features, and other advanced features. The essence of the module, though, is a process which loops through receiving messages from its mailbox, dispatching the requests to callback functions in another module, and sending replies.

```
-module(gen_server_lite).
-export([start/2,do_start/2]).
-export([call/2,cast/2]).

call(Server, Msg) ->
 Ref = make_ref(),
 Server ! {'$call', {Ref, self()}, Msg},
 receive
  {Ref, Reply} -> Reply
 end.

cast(Server, Msg) ->
 Server ! {'$cast', self(), Msg}.

do_start(Module, InitArg) ->
 {ok, State} = Module:init(InitArg),
 loop(Module, State).

loop(Module, State) ->
 receive
  {'$call', {Ref, Who} = From, Msg} ->
   {reply, Reply, NewState} = Module:handle_call(Msg, From, State),
   Who ! {Ref, Reply},
   loop(Module, NewState);
  {'$cast', From, Msg} ->
   {noreply, NewState} = Module:handle_cast(Msg, From, State),
   loop(Module, NewState)
 end.

start(Module, InitArg) ->
 spawn(?MODULE, do_start, [Module, InitArg]).
```

Listing 18: Lightweight implementation of generic gen_server code

With this in mind we will create a lightweight model of gen_server to show the essence of the changes necessary to perform lightweight type checking *and then* perform the same modifications to the real gen_server library. This will allow us to discuss the principles of runtime verification in isolation from the intricacies of the real implementation.

Listing 18 contains the entire implementation of the lightweight model. The start function spawns the server process which first initialises itself using the callback module's init function before looping. When an incoming message arrives it dispatches the request in the message to the corresponding callback function, sends a reply if necessary, then loops again.

The state machine in figure 8.5 on the following page shows how the server behaves: when it receives a cast message it calls the handle_cast callback function and loops again, and when it receives a call message it runs the handle_call callback, sends a response, and then loops.

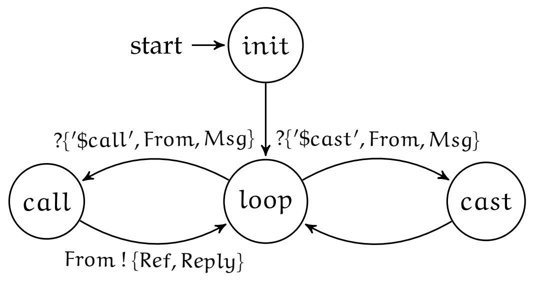


Figure 8.5: State machine model of `gen_server_lite`

```
Data: Core Erlang module definition
Result: inferred message types for known callbacks
if module implements an OTP behaviour then
    for function in module do
        if function is a known callback then
            determine which argument corresponds to message;
            for clause in function do
                infer type of message argument;
            end
            create a list of all inferred types for function;
        end
    end
end
```

Algorithm 9: Type metadata collection algorithm for Core Erlang modules

This lightweight model is enough to run our original counter:

```
1> Server = gen_server_lite:start(my_counter_gen, []).
<0.84.0>
2> gen_server_lite:cast(Server, {add, 20}).
ok
3> gen_server_lite:cast(Server, {sub, 5}).
ok
4> gen_server_lite:call(Server, get).
15
```

8.4.5 Callback Type Inference

The generic Erlang/OTP behaviours rely on a separation of concern where the generic boilerplate code is separated from the specific application code. These two halves interact with each other via *callback* functions written in the specific part. For example, the minimal set of callbacks required to implement a complete `gen_server` process are `init/1`, `handle_cast/2`, and `handle_call/3`.

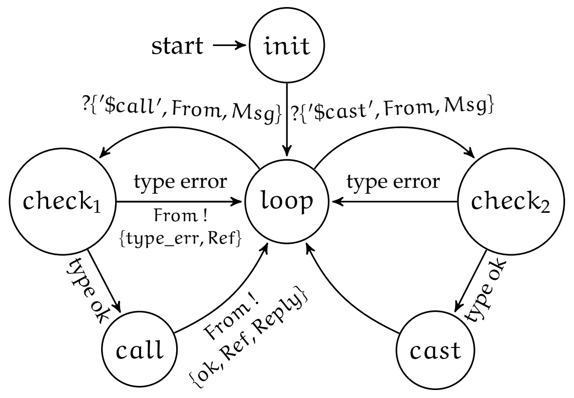


Figure 8.6: State machine model of `gen_server_lite` with type checking

The two functions with the prefix `handle_` are responsible for handling and responding to incoming requests. In all cases the first argument is the request made by the client.

We infer the types of requests accepted by the server using the type inference functions in section 8.4.2. Algorithm 9 on the previous page shows how we do this:

1. Check whether the module has a `behaviour` attribute containing the name of a generic OTP behaviour such as `gen_server`.
2. If so, determine which argument of each callback function corresponds to the client request.
3. Infer the accepted type of requests using `clause_type`.
4. Collect all inferred types together into a list.

The result is a list of inferred message types for each callback function. For example the inferred callback types for `handle_cast` in version 1 of the counter are:

[{`'add'`,term()}, {`'sub'`,term()}]

8.4.6 Analysing Incoming Messages

To analyse the types of incoming messages we add intermediate states to the lightweight server between receiving a request and dispatching to the callbacks, as seen in figure 8.6. After receiving the message its type will be checked against those inferred at compile time. If the message is a sub-type of one of the inferred types then the request will proceed as normal, but if the message is not a sub-type then the callback will never be called and the server goes back to waiting for another message. Additionally, if the incoming request is a *call* the client will expect a response, so we will return an error message to the client when the type check fails.

```
-module(gen_server_lite_typed).
-export([start/2,do_start/2]).
-export([call/2,cast/2]).

call(Server, Msg) ->
 Ref = make_ref(),
 Server ! {'$call', {Ref, self()}, Msg},
 receive
  {type_error, Ref} -> {error, type_error};
  {ok, Ref, Reply} -> {ok, Reply}
 end.

cast(Server, Msg) ->
 Server ! {'$cast', self(), Msg}.

do_start(Module, InitArg) ->
 {ok, State} = Module:init(InitArg),
 loop(Module, State).

loop(Module, State) ->
 receive
  {'$call', {Ref, Who} = From, Msg} ->
   MsgType = types:type_of(Msg), % get type of message
   Types = Module:type_info({handle_call, 3}), % retrieve type info
   case types:is_subtype_list(MsgType, Types) of % check sub-typing
    true ->
     {reply, Reply, NewState} = Module:handle_call(Msg, From, State),
     Who ! {ok, Ref, Reply},
     loop(Module, NewState);
    false ->
     Who ! {type_error, Ref},
     loop(Module, State)
  end;
  {'$cast', From, Msg} ->
   MsgType = types:type_of(Msg), % get type of message
   Types = Module:type_info({handle_cast, 2}), % retrieve type info
   case typse:is_subtype_list(MsgType, Types) of % check sub-typing
     true ->
      {noreply, NewState} = Module:handle_cast(Msg, From, State),
      loop(Module, NewState);
     false ->
      loop(Module, State)
   end
 end.

start(Module, InitArg) ->
 spawn(?MODULE, do_start, [Module, InitArg]).
```

Listing 19: Lightweight implementation of generic gen_server code with type checking

The full implementation of this new model is shown in listing 19 on the previous page, which has a few differences to the code in listing 18 to perform type checking. The first change is that the message format for calls has changed: the response is now either a type error tuple, or an "ok" tuple containing the reply from the `handle_call` function. On the server side we first get the type of the message (lines 23 and 35), retrieve the injected type information relating to the callback function (lines 24 and 36), and check whether the sub-typing relation holds for any of the inferred callback types (lines 25 and 37).

When we try to reproduce the race condition and crash we saw with our counter using the parse transform and our lightweight `gen_server` model we see it is now protected from the malformed request and doesn't crash:

```
1> code:load_file(my_counter_gen), beam_lib:version(my_counter_gen).
{ok,{my_counter_gen,"1"}}
2> Server = gen_server_lite_typed:start(my_counter_gen, []).
<0.85.0>
3> gen_server_lite_typed:cast(Server, {add, 20}).
ok
4> code:load_file(my_counter_gen), beam_lib:version(my_counter_gen).
{ok,{my_counter_gen,"2"}}
5> gen_server_lite_typed:cast(Server, {sub, 5}).
ok
6> gen_server_lite_typed:call(Server, get).
20
```

When the second version of the counter module is loaded the server automatically starts using the new callback functions. Therefore when we send the *old* `sub` message after the reload the type check fails and the callback is never called, so the server is still running when we call it to request its state.

8.4.7 Extending the Real gen_server Module

The lightweight version of the `gen_server` in listing 19 protects server processes from communication discrepancies by performing runtime type checking. One shortcoming of this model is due to the way the `cast` function behaves: it is a fire-and-forget request without a response from the server. As there is no response from the server any communication discrepancies in casts are handled silently, simply returning the atom `'ok'` to indicate that the message was sent to the server process.

We address this shortcoming in the modification of the real `gen_server` library by adding logging information about type errors. The library consists of several hundred lines of codes and many modifications are required to data structures and function definitions in order to propagate type information, handle exceptions, and interact with the virtual machine. At a high level however, the process is the same as with our lightweight model: add type checks at the point where messages are received and only run callback functions if the request's type is a sub-type of the inferred types.

When we combine the type inference, metadata injection, and modifications to the `gen_server` library we create a runtime verification system which protects servers from communication discrepancies and produces error reports using the standard library's logging mechanisms:

```
1> {ok, Server} = my_counter_gen:start(0).
{ok,<0.85.0>}
2> gen_server:cast(Server, {add, 20}).
ok
3> gen_server:cast(Server, {op, sub, 5}).
ok
=ERROR REPORT====
** Message with incompatible type for my_counter_gen:handle_cast/3 received
** Message with incompatible type: {op, sub, 5}
** Message type: {tuple,[{literal,op,atom},{literal,sub,atom},{literal,5,integer}]}
4> gen_server:call(Server, get).
20
```

The server has rejected the incompatible request, generated an error report, and continued to run.

This runtime verification process is entirely automatic: no input from the user is required, type inference and metadata injection occur as part of the compilation process, and the runtime type checking is embedded in the generic portion of the `gen_server` library.

The approach combines the strengths of static analysis and runtime verification by performing type inference at compile time and then using the sub-typing relation at runtime to increase the robustness of the client-server interactions. Furthermore, this approach allows Erlang programmers to remove boilerplate "catch-all" clauses from their `receive` expressions and callback functions with the knowledge that the typechecking infrastructure in the `gen_server` library will intercept incompatible requests before they crash the server.

Chapter 9

Related Work

Erlang was created to operate in the presence of software errors: the creators of the language argue that bugs are a "fact of life", and that programmers should instead build *fault-tolerant* applications by using *lightweight processes* with *strong isolation*.

Unfortunately, not all software errors are created equal: some errors in Erlang programs may emanate from external sources (such as hardware devices or network interfaces) and are best suited to being dealt with at runtime by a well-designed application, but other errors are easily detectable (and preventable) at compile time or runtime. As a result, there have been many efforts over the years to make Erlang a "safer" language through a variety of means: formal models of the language and its runtime to reason about how specific programs behave, static analysis tools to detect data type errors, and runtime verification tools to protect applications from system states which violate some user-defined safety properties. This work has not been carried out in isolation from the larger academic and industrial communities, however: there is a significant overlap between techniques used to analyse Erlang programs and the techniques used with other formal models and real-world programming languages.

In this part of the thesis I explore the work most closely related to that presented in previous chapters: formal models and type systems for Erlang, techniques for modelling and reasoning about Erlang's message passing system, existing software tooling for static analysis and runtime verification.

9.1 Formal Models

The most concrete definition of Erlang is the *Core Erlang 1.0.3 Language Specification* (Carlsson et al. 2004): a written specification of the Core Erlang intermediate format. In chapter 4 we formalised a *fragment* of the Core Erlang specification in a language named CoErl. The most important parts relating to communication were formalised there: pattern matching, guard evaluation, mailboxes, and the send/receive operations. Some liberties were taken in this formalisation: the specification does not specify an evaluation order (so left-to-right was chosen in line with the behaviour of Erlang/OTP), and the task of sending messages is explicitly omitted (again, the behaviour of Erlang/OTP was used). The result of this formalisation was a fragment of the full Core Erlang language, notably without any exception handling or higher-order behaviour. Regardless, the model fulfils its purpose: serving as a minimal model of Erlang's communication for non-trivial programs.

Perhaps the most complete formal model of Erlang is the small-step operational semantics for a the Erlang-F (*Erlang F*ragment) language behind the **McErlang** model checker (L. Fredlund and Svensson 2007). The language was based on a version of Erlang which predated the creation of the Core Erlang intermediate format. Erlang-F – like CoErl presented in chapter 4 – represents a communicating fragment of the Erlang programming language. Patterns, guards, and mailbox behaviour are all implemented in Erlang-F. This tool is based heavily on the PhD thesis of one of the authors which explores an operational semantics for Erlang (L.-Å. Fredlund 2001). Comparing the operational semantics from chapter 4 with this thesis we note many similarities: we have chosen a left-to-right evaluation order in the absence of any formal specification and we have focused on a communicating subset of Erlang's syntax to avoid large amounts of syntactic sugar. L.-Å. Fredlund notably models non-local control flow via exceptions, which we have not formalised.

There are also resemblances between Erlang and other formal systems, notably **CSP** and the **π-calculus**. Communicating Sequential Processes (CSP) is a mathematical model (and arguably a programming language) for concurrent process-oriented programming similar in vein to Erlang: lightweight processes send and receive messages to oneanother, and communication can be directed by "guarded expressions" (Hoare 1985). On the other hand, the π-calculus is a minimalist process calculus where messages are explicitly sent and received over dedicated (and named) communication channels (Milner, Parrow and Walker 1992). There are significant differences between these models and the implementation of Erlang, however: while CSP is effectively a "usable" programming language it lacks some of the more powerful features of Erlang's mailbox behaviour, and the portability of channels in the π-calculus cannot be fully realised in Erlang as mailboxes are attached to processes and cannot be moved. Despite these differences the two systems serve as a useful "common denominator": many Erlang programs can be written in the π-calculus, for example (Noll and Roy 2005), allowing existing analysis techniques to be used for reasoning about the behaviour of Erlang programs.

9.2 Model Checking

In chapter 5 we analysed the communicating behaviour of CoErl via a labelled version of the small-step operational semantics from chapter 4. Each transition in the system was given one of four different labels: send $(\iota \,!\, m)$, receive $(?\, m)$, arrive $(\mathit{arr}\,)$, or internal (τ).

This model is known as a *labelled transition system* and serves as the foundation for many analyses of concurrent systems: create a trace of the execution of a system and then analyse that trace to detect software errors or to check safety properties. The previously mentioned McErlang model checker operates in part using this technique: it traces the execution of processes under different schedulings to ensure that the system satisfies properties specified by a model.

Concuerror (Gotovos, Christakis and Konstantinos Sagonas 2011) is a tool which operates in a similar vein: it takes as inputs an Erlang program and a model of its behaviour and runs it under systematically chosen schedulings in order to elicit any race conditions or other concurrency-related software defects. This analysis is a sound over-approximation: it over-approximates the behaviour of programs, but proves that all behaviours it explores are safe.

The main difference between the labelled transition system from chapter 4 and these two tools is that the former serves solely as a useful model to reason about Erlang's communication in isolation from other program behaviour, while the latter are used to trace the execution of real systems at runtime.

In addition, the **Soter** tool derives a finitely representable over-approximation of Erlang programs (D'Osualdo, Kochems and Ong 2013). It takes an Erlang module, a specification of a safety property, generates a Petri net which represents an abstract model of the Erlang module, and then calls the BFC solver (Kaiser, Kroening and Wahl 2014).

There are many other approaches to model checking, many of them now considered standard (Clarke et al. 2018). We could use a temporal logic to reason about the communicating behaviour of Erlang programs (though it would be difficult to represent out-of-order communications), use a Petri net model of Erlang programs to approximate the behaviour of mailboxes. Using one of these model checking techniques we would likely be able to detect more communication discrepancies automatically as we would be able to reason about the order of events and causality between communications, which we currently lack. By contrast, our current type-based analysis is already capable of detecting common communication discrepancies due to the relatively common use of patterns and guards in Erlang programs.

Another way of verifying the behaviour of a program is symbolic execution. There has been work to reason about the behaviour of Erlang programs using term rewriting and a technique called narrowing, allowing the behaviour of an Erlang program – including non-deterministic concurrent behaviour – to be over-approximated without executing it (Vidal 2013). The work has been iterated upon, allowing for both an under-approximation and over-approximation of Erlang programs (Vidal 2014). It is the first known attempt to formalise symbolic execution of Erlang programs which deals with symbolic data, not just possible schedulings. A concrete semantics for an Erlang fragment is also introduced: it has similar properties to CoErl and the small-step semantics in chapter 6. In addition, Vidal makes scheduling an explicit part of the semantics, whereas in CoErl it is implicit via the non-determinism of the concurrent small-step relation. Ultimately, this adventure into symbolic execution for Erlang has resulted in a causal-consistent **replay-based debugger** for Erlang programs which allows users to record executions of a program and replay only the actions which led to a specific behaviour (Lanese, Palacios and Vidal 2019). As part of the analysis a labelled "logging semantics" is used which has send and receive labels which are similar to those seen in CoErl. The main difference however is that due to the explicit nature of scheduling, a *global* mailbox is used to co-ordinate message delivery between processes, which is a stark contrast to the ad-hoc communication used in CoErl's labelled small-step semantics.

9.3 Behavioural Types

Behavioural types are a technique for describing the behaviour of software using a type system, in contrast to the typical use of describing data (Gay and Ravara 2017). **Session types** are a popular form of behavioural contract which describes the behaviour of a communicating system using protocols: a specification which describes the way two or more concurrent components may communicate with each other. Session types are used in various ways: they can be used as documentation, for static analysis (where the implementation of a protocol is checked against its specification), runtime verification (where compliance with a protocol is checked at runtime), or using a combination thereof.

There have been several adaptations of session types for Erlang. First, session types were introduced to a "featherweight" version of Erlang with restrictions on the communication model (Mostrous and Vasconcelos 2011). This was followed by an implementation of *multiparty session actors* where generic server processes are viewed as actors and compliance with a session type can be verified at runtime (Fowler 2016). Later work introduced an adapted OTP behaviour which used a combination of static analysis and a sound recovery algorithm to automatically determine the compile-time dependencies between processes and generate restart strategies for crashed processes (Neykova and Yoshida 2017).

One area which session types often struggle to address is Erlang's *out-of-order* mailbox behaviour: while it is possible to extend session types to asynchronous communication and even model timeouts (Bocchi et al. 2019), session types often do not model Erlang's ability to selectively receive messages in a different order to which they arrived. Specifically, session types rely on *protocol* definitions where each send by one party corresponds with a receive by the other, and vice versa. Instead, the static and runtime analysis methods presented in chapter 8 focus on detecting real discrepancies instead of forcing a strict communication protocol on all parties. Furthermore, it is not clear how a session types would apply to the "open world" model of Erlang applications wherein any process can send a message to another process knowing only its PID: no explicit communication channel needs to be established in advance.

9.4 Testing & Fault Injection

Model checking tools are exhaustive: they analyse all possible executions of a program to prove that some specified properties hold. The aforementioned McErlang and Concuerror tools verify properties of Erlang programs by running them, analysing how they behave, and then forcing specific schedulings to explore the state space of programs. Unfortunately, model checking can take a significant amount of time to verify properties (minutes, hours, or more) based on the size or complexity of programs and properties.

Instead, many Erlang programmers use non-exhaustive testing to give themselves reasonable confidence that their programs behave as expected. This can take the form of hand-written unit and integration tests which use the **EUnit** or **Common Test** libraries which ship with Erlang/OTP, or via other third party packages. One popular testing library is **QuickCheck** (QuviqQ A.B. 2019) which performs random *property based testing* wherein user-defined properties are analysed to automatically generate and run random test cases (Claessen and Hughes 2000). The library has been used to test (and find bugs in) Erlang telecom software (Arts, Hughes and Johansson 2006) and find race conditions in concurrent programs (Claessen, Palka, Smallbone et al. 2009). Furthermore, property based testing can be used to test models of finite state machines written in Erlang (Seijas 2017).

Another way of analysing concurrent (and even distributed) Erlang programs is via *fault injection*: deliberately "losing" messages between networked nodes, crashing critical processes, deliberately slowing the execution of a process by manipulating the scheduler, and so forth. The aim of this kind of analysis is to examine how applications deal with error conditions in real-world scenarios. The **Partisan** library (Meiklejohn 2018) is an extensive suite of software which can be used to test Erlang applications in this way: it allows users to inject faults into running applications, trace the execution of entire distributed systems, and also implements a new distribution protocol for Erlang/OTP aimed at alleviating common issues.

9.5 Type Systems

Although Erlang is a *dynamically typed* programming language the compiler still performs some cursory type checks. For example, it can detect when non-numeric arguments are passed to arithmetic operators if one of the operands can be resolved to a constant value at compile time, and it can determine that clauses are dead if they appear after a catch-all. Despite many improvements to the Erlang compiler during its lifetime, programs can still contain bugs that would be easily detectable with a static type system, for example incorrect types of function arguments.

There have been attempts to add a static type system to Erlang, however. One of the earliest endeavours was a practical sub-typing system which supported variable types, clauses, function types, and even a simple analysis of message types (Marlow and Wadler 1997). This analysis was performed on Erlang itself, as it predated the creation of Core Erlang; later work investigated a type system for the functional aspect of Core Erlang (Nyström 2003).

Although its stated purpose is to detect *discrepancies* in Erlang software, the **Dialyzer** (Konstantinos Sagonas 2005) tool performs static type checking of Erlang programs using *success typings* which approximates the type of variables and functions based on how they are used (Lindahl and Konstantinos Sagonas 2006). The logic used in Dialyzer has also been extended to analysing asynchronous communication: certain race conditions and orphan messages can be detected at compile time (Christakis and Konstantinos Sagonas 2010). More recently, the **Gradualiser** tool has brought *gradual typing* to Erlang, allowing parts of a program to be statically typed while others remain dynamically typed. Gradualiser distinguishes between Erlang's $\mathtt{any}()$ type and $\mathtt{term}()$ type: $\mathtt{any}()$ is used for *dynamically typed* portions of a program, while $\mathtt{term}()$ is used for *statically typed* portion. The tool explicitly does not perform type inference on entire programs: it checks whether the provided type specifications match the implementation.

The project is an effort to create the performance of Erlang systems by automatically compiling parts of a program to native machine code (Kostis Sagonas et al. 1998). The project is closely related to Core Erlang, which it uses to reason about the behaviour of variables and function applications. Erlang modules compiled with benefit from the speed of optimised native code, but frequent context switches between BEAM bytecode and native code can cause performance issues.

In chapter 6 a sub-typing system was created based on Erlang's existing type system and the behaviour of CoErl, which in turn is based upon Core Erlang. This relates our work to the HiPE project via the use of Core Erlang, but the two systems have different goals: HiPE is used for optimisation purposes and is rather conservative (backing out when it cannot determine a type), while the type system in chapter 6 and the analysis from chapter 8 specifically seek these discrepancies. On the other hand, Gradualizer is a promising project with potential for expansion: our sub-typing relation, guard type inference techniques, and concepts of message compatibility could be used to improve the quality of Gradualizer's type checking, or be used in our analysis to provide data flow information or type information about variables.

9.6 Runtime Analysis & Profiling

To aid in debugging the BEAM has a plethora of built-in diagnostic tools which capture runtime information about how an Erlang system behaves: scheduler information, process memory usage, execution tracing, and inter-process communication to name a few. The **Percept2** tool amalgamates several of these information sources into a single tool, allowing Erlang developers to view information about their systems both online and offline (Li and Simon J. Thompson 2013).

These APIs are also used by other tools to perform *runtime verification*: extracting information from a running system and verifying properties of that system. For example, the **ELarva** tool (Colombo, Francalanza and Gatt 2012) uses online tracing to verify that safety-critical and security-critical processes remain compliant with user specified security properties. The **detectEr** library (Cassar and Francalanza 2015) operates on a similar principle: it verifies the correctness of Erlang processes at runtime using automatically generated monitoring code which can monitor processes either synchronously or asynchronously via automatic instrumentation.

Runtime monitoring tools are motivated by the non-exhaustiveness of test suites and the long running times of model checkers: it is often quicker (and satisfactory) to defer verification to runtime in exchange for a relatively small performance loss caused by the overhead of monitoring.

Chapter 10

Conclusions

In chapter 1 we identified a potential issue with concurrent Erlang programs where mismatches between the way processes send and receive messages can cause memory leaks and system crashes. This thesis has presented techniques for reasoning about, detecting, and mitigating against these communication discrepancies:

- **An operational semantics for a communicating fragment of Core Erlang** was used to reason about the behaviour of Erlang processes (chapter 4). In order to reason about mailboxes in isolation from other computation, chapter 5 presented a labelled transition system and trace semantics for Erlang communications.

- **A semantic sub-typing system for Erlang** was created based on Erlang's semantics, with union, intersection, and negation types enabling us to reason about the types of values that patterns, guard expressions, and clauses will accept chapter 6. This was accompanied by an original BDD based sub-typing algorithm which served as a stepping stone to an implementation chapter 7.

- **A combination of static type inference and runtime verification to automatically detect communication discrepancies** both at compile time and runtime chapter 8. The static type inference can be used to detect *definite* discrepancies at compile time while the runtime verification enables discrepancies to be detected while programs are executing, protecting them from crashes.

This thesis has presented a formalisation of a communicating Core Erlang which served as the foundation for a communications analysis of Erlang programs. The model was then used to create a sub-typing system which can be used to approximate the behaviour of mailboxes which was then used to create a fully automatic hybrid verification system which can detect real communications discrepancies in Erlang programs.

The objective of this analysis is not to change the way Erlang programmers write their programs or force them into a specific programming style, but rather to increase the confidence they have in the way their programs communicate, avoiding memory leaks and preventing server processes from malformed requests which would otherwise cause them to crash.

We believe that there is a role for lightweight tools for permissive languages such as Erlang. The expressive syntax of Erlang allows programmers to rapidly develop large and sophisticated concurrent applications by leveraging Erlang's unique communication model. Without the restrictions of a static type system or any requirement of proving the safety of communications, Erlang programmers are free to use complex programming techniques. We also argue that tools for languages such as Erlang are more valuable than tools for less permissive languages as there is more room for error in the absence of a dependent type system or notion of checked exceptions, for example. Writing tools for languages like Erlang enables programmes to inspect legacy code without

10.1 Future Work

There are several opportunities for future work that would expand upon the ideas presented in earlier chapters. Specifically, a desirable goal would be to create a useful software tool for Erlang developers which automatically analyses and checks communications in their applications to protect them from discrepancies. With this in mind, the following are some ideas to increase the scope and impact of the work:

Formalisation The operational semantics for CoErl in chapter 4 only capture the core concepts of Erlang: patterns, guards, processes, and communication. The model is lacking non-local control flow (via Erlang's error and exception handling mechanisms) and higher-order behaviour. It would be interesting for formalise a larger part of the Core Erlang specification in order to understand how these features affect mailbox behaviour, and to discover whether there is room for improvement on the specification.

Type System The type system and sub-typing relation in chapter 6 contains a representative subset of Erlang's built-in types: atoms, numeric types, PIDs, lists, and tuples. One discrepancy between our system and Erlang's existing *type notation* is *proper lists*: we view lists strictly as improper (i.e. a list is either a cons cell or the empty list) while Erlang supports a syntax for proper lists (i.e. those which are nil-terminated). Future work could bring our type system into alignment with Erlang's notation, possibly extending the type system to support map and binary types in the process.

Type Inference & Static Analysis The type inference algorithms presented in section 6.4 and the implementation offered in chapter 8 reason about the types of patterns, guards, and clauses. Furthermore, no data flow analysis is performed. This naturally leads to an over-approximation of the types of variables found in patterns and guards as we have no type information for them. It would be useful to integrate with an existing type inference or type checking tool for Erlang such as Dialyzer or Gradualiser to provide type information, or perhaps to improve the type inference mechanisms of those tools. In addition, the current BDD based sub-typing algorithm is only a proof-of-concept: it does not garbage collect orphan nodes in the graph and it makes basic use of Erlang's *Erlang Term Storage* (ETS) tables for memoisation purposes. Further work could look at improving the efficiency of the BDD representation and construction algorithms, or using a different technique to represent types and decide the sub-typing relationship.

Runtime Verification Finally, the current approach to runtime verification in chapter 8 operates by receiving messages from a process' mailbox, checking their types, and passing them on to callback functions only if their types are deemed compatible. This is a proof of concept of the practicality of performing type checking at runtime, but it effectively serialises the mailbox, preventing out-of-order communication. Instead, type checking could be integrated with the ERTS by modifying the BEAM to perform type checking of messages "in flight", ensuring that messages are not placed in mailboxes if they will never be received.

Bibliography

Arts, Thomas, John Hughes and Joakim Johansson (2006). 'Testing telecoms software with quviq QuickCheck'. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang - ERLANG '06*. ACM Press. DOI: 10.1145/1159789.1159792.

Bevemyr, Johan (2018). 'How Cisco is using Erlang for intent-based networking'. Code BEAM STO.

Bocchi, Laura et al. (2019). 'Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes'. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Vol. 11423. Lecture Notes in Computer Science. Springer, pp. 583–610. DOI: 10.1007/978-3-030-17184-1_21.

Brace, Karl S., Richard L. Rudell and Randal E. Bryant (1990). 'Efficient Implementation of a BDD Package'. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*. IEEE Computer Society Press, pp. 40–45. DOI: 10.1145/123186.123222.

Carlsson, Richard (2001). 'An introduction to Core Erlang'. In: *In Proceedings of the PLI'01 Erlang Workshop*.

— (Mar. 2019). *cerl.erl, Erlang/OTP compiler application*. URL: https://github.com/erlang/otp/blob/41672f4/lib/compiler/src/cerl.erl.

Carlsson, Richard et al. (Nov. 2004). *Core Erlang 1.0.3 Language Specification*. Tech. rep. Uppsala University. URL: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

Cassar, Ian and Adrian Francalanza (Feb. 2015). 'On Synchronous and Asynchronous Monitor Instrumentation for Actor-based systems'. In: *Electronic Proceedings in Theoretical Computer Science* 175, pp. 54–68. DOI: 10.4204/eptcs.175.4.

Christakis, Maria and Konstantinos Sagonas (2010). 'Static Detection of Race Conditions in Erlang'. In: *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, pp. 119–133. DOI: `10.1007/978-3-642-11503-5_11`.

Claessen, Koen and John Hughes (2000). 'QuickCheck: a lightweight tool for random testing of Haskell programs'. In: *ICFP*. ACM, pp. 268–279.

Claessen, Koen, Michal Palka, Nicholas Smallbone et al. (2009). 'Finding race conditions in Erlang with QuickCheck and PULSE'. In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09*. ACM Press. DOI: `10.1145/1596550.1596574`.

Clarke, Edmund M. et al. (Dec. 2018). *Model Checking, Second Edition*. MIT Press.

Colombo, Christian, Adrian Francalanza and Rudolph Gatt (2012). 'Elarva: A Monitoring Tool for Erlang'. In: *Runtime Verification*. Springer Berlin Heidelberg, pp. 370–374. DOI: `10.1007/978-3-642-29860-8_29`.

D'Osualdo, Emanuele, Jonathan Kochems and Luke Ong (2013). *SOTER - Safety verifier fOr The ERlang language*. URL: `https://mjolnir.cs.ox.ac.uk/soter/` (visited on 15/06/2018).

Erlang/OTP Team (2018). *Erlang Reference Manual*. v9.3. Ericsson A.B.

— (2019a). *compile: Erlang compiler*. Ericsson A.B. URL: `https://erlang.org/doc/man/compile.html`.

— (2019b). *Efficiency Guide*. 10.3. Ericsson A.B. URL: `http://erlang.org/doc/efficiency_guide/introduction.html`.

— (2019c). *Erlang/OTP Documentation*. Ericsson A.B. URL: `http://erlang.org/doc/`.

— (2019d). *OTP Design Principles User's Guide*. 10.3. Ericsson A.B. URL: `http://erlang.org/doc/design_principles/users_guide.html`.

Facebook Inc. (Jan. 2018). *Facebook Reports Fourth Quarter and Full Year 2017 Results*.

Fowler, Simon (2016). 'An Erlang Implementation of Multiparty Session Actors'. In: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*. Vol. 223. EPTCS, pp. 36–50. DOI: `10.4204/EPTCS.223.3`.

Fredlund, Lars-Åke (2001). 'A framework for reasoning about Erlang code'. PhD thesis. Mikroelektronik och informationsteknik.

Fredlund, Lars-Åke and Hans Svensson (2007). ‘McErlang: A Model Checker for a Distributed Functional Programming Language’. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’07. Freiburg, Germany: ACM, pp. 125–136. ISBN: 978-1-59593-815-2. DOI: `10.1145/1291151.1291171`.

Gay, Simon and António Ravara (2017). *Behavioural Types: from Theory to Tools*. River Publishers.

Gotovos, Alkis, Maria Christakis and Konstantinos Sagonas (2011). ‘Test-Driven Development of Concurrent Programs Using Concuerror’. In: *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. Erlang ’11. Tokyo, Japan: ACM, pp. 51–61. ISBN: 9781450308595. DOI: `10.1145/2034654.2034664`.

Harrison, Joseph (2017). ‘Towards an Isabelle/HOL Formalisation of Core Erlang’. In: *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang, Oxford, United Kingdom, September 3-9, 2017*. ACM, pp. 55–63. DOI: `10.1145/3123569.3123576`.

— (2018). ‘Automatic Detection of Core Erlang Message Passing Errors’. In: *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, ICFP 2018, St. Louis, MO, USA, September 23-29, 2018*. ACM, pp. 37–48. DOI: `10.1145/3239332.3242765`.

— (2019). ‘Runtime Type Safety for Erlang/OTP Behaviours’. In: *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2019, Berlin, Germany, August 18, 2019*. ACM, pp. 36–47. DOI: `10.1145/3331542.3342571`.

Hebert, Fred (2013). *Learn You Some Erlang for Great Good!: A Beginner’s Guide*. No Starch Press. ISBN: 9781593274351. URL: `https://learnyousomeerlang.com/`.

Hennessy, Matthew and Robin Milner (1985). ‘Algebraic Laws for Nondeterminism and Concurrency’. In: *J. ACM* 32.1, pp. 137–161.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall. ISBN: 0-13-153271-5.

Huth, Michael and Mark Dermot Ryan (2004). *Logic in computer science - modelling and reasoning about systems (2. ed.)* Cambridge University Press.

Kaiser, Alexander, Daniel Kroening and Thomas Wahl (2014). ‘A widening approach to multithreaded program verification’. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.4, p. 14.

Klophaus, Rusty (2010). ‘Riak core: Building distributed applications without shared state’. In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, p. 14.

Lanese, Ivan, Adrián Palacios and Germán Vidal (2019). 'Causal-Consistent Replay Debugging for Message Passing Programs'. In: *FORTE*. Vol. 11535. Lecture Notes in Computer Science. Springer, pp. 167–184.

Li, Huiqing and Simon J. Thompson (2013). 'Multicore profiling for Erlang programs using Percept2'. In: *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, Massachusetts, USA, September 28, 2013*. ACM, pp. 33–42. doi: `10.1145/2505305.2505311`.

Lindahl, Tobias and Konstantinos Sagonas (2006). 'Practical type inference based on success typings'. In: *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*. ACM, pp. 167–178. doi: `10.1145/1140335.1140356`.

Marlow, Simon and Philip Wadler (1997). 'A Practical Subtyping System for Erlang'. In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP '97. Amsterdam, The Netherlands: ACM, pp. 136–149. isbn: 0-89791-918-1. doi: `10.1145/258948.258962`.

Meiklejohn, Christopher S. (2018). 'Partisan: Enabling Real-World Protocol Evaluation'. In: *Proceedings of the 2018 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed systems, ApPLIED@PODC 2018, Egham, United Kingdom, July 27, 2018*. ACM, pp. 45–48. doi: `10.1145/3231104.3231106`.

Milner, Robin (1980). *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer.

Milner, Robin, Joachim Parrow and David Walker (1992). 'A Calculus of Mobile Processes, I'. In: *Inf. Comput.* 100.1, pp. 1–40. doi: `10.1016/0890-5401(92)90008-4`.

Mostrous, Dimitris and Vasco Vasconcelos (2011). 'Session Typing for a Featherweight Erlang'. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 95–109. doi: `10.1007/978-3-642-21464-6_7`.

Neykova, Rumyana and Nobuko Yoshida (2017). 'Let it recover: multiparty protocol-induced recovery'. In: *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*. ACM, pp. 98–108. url: `http://dl.acm.org/citation.cfm?id=3033031`.

Nicola, Rocco De (1987). 'Extensional Equivalences for Transition Systems'. In: *Acta Informatica* 24.2, pp. 211–237.

Noll, Thomas and Chanchal Kumar Roy (2005). 'Modeling Erlang in the pi-calculus'. In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, Tallinn, Estonia, September 26-28, 2005*. ACM, pp. 72–77. DOI: `10.1145/1088361.1088375`.

Nyström, Sven-Olof (2003). 'A Soft-typing System for Erlang'. In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*. ERLANG '03. Uppsala, Sweden: ACM, pp. 56–71. ISBN: 1-58113-772-9. DOI: `10.1145/940880.940888`.

Pearce, David J. (2013). 'Sound and Complete Flow Typing with Unions, Intersections and Negations'. In: *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 7737*. VMCAI 2013. Rome, Italy: Springer-Verlag, pp. 335–354. ISBN: 978-3-642-35872-2. DOI: `10.1007/978-3-642-35873-9_21`.

Pivotal Software (2019). *RabbitMQ open source message broker*. URL: `https://www.rabbitmq.com/`.

QuviqQ A.B. (2019). *QuviQ QuickCheck*. URL: `http://www.quviq.com/products/erlang-quickcheck/`.

Roscoe, Bill (1998). *The theory and practice of concurrency*. ISBN: 978-0-13-674409-2.

Sagonas, Konstantinos (2005). 'Experience from developing the Dialyzer: A static analysis tool detecting defects in Erlang applications'. In: *Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools*.

Sagonas, Kostis et al. (1998). *The High-Performance Erlang Project*. URL: `https://www.it.uu.se/research/group/hipe/`.

Seijas, Pablo Lamela (2017). 'Model construction, evolution, and use in testing of software systems'. PhD thesis. University of Kent, Canterbury, UK. URL: `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.754827`.

Shannon, Claude E (1949). 'The synthesis of two-terminal switching circuits'. In: *The Bell System Technical Journal* 28.1, pp. 59–98.

Thompson, Simon J and Dominic A Orchard (Jan. 2019). *CO545: Functional and Concurrent Programming*. University of Kent.

Vidal, Germán (2013). 'Towards Erlang Verification by Term Rewriting'. In: *LOPSTR*. Vol. 8901. Lecture Notes in Computer Science. Springer, pp. 109–126.

— (2014). 'Towards Symbolic Execution in Erlang'. In: *Ershov Memorial Conference*. Vol. 8974. Lecture Notes in Computer Science. Springer, pp. 351–360.

Glossary

CoErl A communicating fragment of Core Erlang. 3, 5, 26, 58–60, 63, 66–69, 72, 74, 82, 83, 86, 90, 92–95, 100, 101, 110, 111, 136, 152, 157, 172–174, 177, 179

Core Erlang An intermediate representation of Erlang used in the Erlang/OTP compiler. 3

Erlang/OTP The de facto distribution of Erlang consisting of the compiler toolchain, virtual machine, runtime system, and supporting libraries. 3, 6, 8, 23, 24, 136, 137, 150, 165, 172

HiPE High Performance Erlang. 152, 176, 177

Acronyms

BDD Binary Decision Diagram. 4, 5, 27–32, 34, 35, 37, 109, 119–124, 126–131, 133–135, 151, 178, 180

BEAM Bogdan/Bjorn's Erlang Abstract Machine. 2, 7, 8, 18, 42, 55, 85, 94, 144, 150, 152, 177, 180

BIF Built-In Function. 15, 16, 96, 99, 110, 120, 156, 158

DAG Directed Acyclic Graph. 33

ERTS Erlang Runtime System. 55, 180

ETS Erlang Term Storage. 151, 180

LTS Labelled Transition System. 6, 27, 66–70, 72, 82, 90, 92

MRDAG Multi-Rooted Directed Acyclic Graph. 32, 33, 35–37, 120–122, 124, 133, 134, 151

OBDD Ordered Binary Decision Diagram. 27, 29–31, 119

OTP Open Telecom Platform. 6, 163, 166

PID Process Identifier. 19, 20, 22, 54, 58, 60, 63, 68

ROBBD Reduced Ordered Binary Decision Diagram. 6, 27, 31–35, 37, 119–121, 127, 131–134

This thesis was created with LuaTeX distributed as part of TeX Live 2019 and was edited with Emacs and Visual Studio Code.
Text is set in TeX Gyre Pagella, source code in `Inconsolata`, and mathematics in `Euler`.
Source code highlighting performed via the minted package which uses the Pygments library.
Graphs created with Ti*k*Z and Graphviz.

Written: 2018–2020
Submitted: January 2020
Viva: May 2020
Corrected: July 2020–September 2020
Awarded: November 2020
Deposited: April 2021