A Semantic Framework for Deterministic Functional Input/Output

A dissertation submitted to the

University of Dublin, Trinity College

for the degree of

Doctor of Philosophy

Malcolm John Dowse

Department of Computer Science

University of Dublin, Trinity College

Ireland

March 2006

Declaration

This thesis has not been submitted as an exercise for a degree at any other university. Except where otherwise stated the work described herein has been carried out by the author alone.

This thesis may be borrowed or copied upon request with the permission of the Librarian, University of Dublin, Trinity College. The copyright belongs jointly to the University of Dublin and the author.

Signed:

Malcolm Dowse

March 12, 2006

Summary

This dissertation presents a pure functional language called Curio. This language is unusual in possessing a rigorous yet general semantics for I/O which permits both formal proofs and a fine-tuned approach to concurrency. This is achieved by describing the language semantics in terms of an abstract model of the I/O application programmer interface (API).

We begin by introducing this model, which simulates how the API affects some global world state and describes the extent to which certain I/O actions are order independent. We then introduce Curio, which extends a pure functional language with five primitives for monadic I/O and explicit concurrency. Each process is at runtime associated with what we call an "I/O context" and this outlines the I/O actions that the process is allowed to perform. The language's semantics are given in terms of a simple implementation in the metalanguage, Core-Clean.

It is proved formally that if an I/O model obeys some broad properties then, despite concurrency, program execution is always deterministic. This result is made more substantial by the fact that concurrent processes may communicate with one another via the API.

We then show how a large fragment of Haskell 98's I/O interface can be modelled in Curio. To do this we found it necessary to develop API combinators which allow larger APIs to be constructed from smaller ones in a way that preserves deterministic behaviour. It was also necessary to solve the technical problem of how two processes may allocate data locally within a single global state. We present some example applications which demonstrate how Curio appears to lead to more expressive, powerful I/O code compared to that of existing approaches.

Finally we tackle the problem of reasoning formally about Curio programs. A notion of program equivalence is developed and we show how the monad laws hold and how equivalence is a congruence. This equivalence relation is powerful because it identifies programs not only by their cumulative effect on global state, but also by how they interact with other processes, thereby allowing us to distinguish non-terminating programs. In the final technical chapter we develop a algebraic structure which describes more succinctly how permissions may be distributed among concurrent processes.

The confluence proof and many other technical results have been machine-verified.

Contents

Acknowledgements **xv**

1 Introduction **1**

1.1 Functional programming and I/O . . . 2
1.1.1 Pure functional languages and I/O . . . 2
1.1.2 Monadic I/O . . . 3
1.1.3 Unique types . . . 4
1.1.4 Other approaches . . . 4
1.2 Semantics of functional languages . . . 5
1.2.1 Introduction to language semantics . . . 5
1.2.2 PCF and applicative bisimulation . . . 6
1.2.3 Proving program properties . . . 6
1.3 Semantics for I/O . . . 7
1.3.1 I/O as state manipulation . . . 7
1.3.2 Bisimulation – the Haskell approach . . . 8
1.3.3 Other approaches . . . 8
1.4 Thesis outline . . . 9
1.4.1 A brief technical introduction to Curio . . . 9
1.4.2 Hypothesis . . . 11
1.4.3 Synopsis . . . 12
1.4.4 Results . . . 13
1.4.5 How to read this dissertation . . . 13
1.4.6 Separate publications . . . 14
1.5 Related work . . . 14
1.5.1 Deterministic concurrency . . . 15
1.5.2 Reasoning about I/O and/or concurrency . . . 16
1.6 Technical preliminaries . . . 17
1.6.1 General prerequisites . . . 17
1.6.2 LCF proof assistants . . . 19

2 I/O models and contexts **21**
2.1 Definition . . . 22
2.1.1 The API – `af` and `wa` . . . 23
2.1.2 I/O Contexts – `ap` . . . 23
2.1.3 Enforcing determinism – `pf` . . . 24
2.2 Five examples . . . 25
2.2.1 Communication buffers . . . 25
2.2.2 Many-to-many mutexes . . . 26
2.2.3 An integer variable . . . 27
2.2.4 Id's I-structures . . . 28
2.2.5 Terminal I/O . . . 29
2.3 Discussion . . . 30
2.4 Chapter summary . . . 32

3 Curio – a language for reasoning about I/O **33**
3.1 Language primitives and examples . . . 34
3.1.1 Communication buffer examples . . . 35
3.1.2 Mutex examples . . . 35
3.1.3 Integer variable examples . . . 35
3.1.4 Terminal I/O examples . . . 36
3.2 Semantics . . . 36
3.3 Convergence/divergence and the implementation . . . 39
3.3.1 Metalanguage encoding . . . 39
3.3.2 Convergence and divergence . . . 41
3.3.3 Convergence is recursively enumerable . . . 42
3.4 Chapter summary . . . 45

4 Confluence **47**
4.1 Introduction . . . 47
4.1.1 Terminology . . . 47
4.1.2 Overview . . . 48
4.2 Preliminaries . . . 49
4.2.1 Deconstructing $\texttt{next}_\texttt{s}$ . . . 49
4.2.2 Annotating single-step reduction . . . 50
4.2.3 A template for inductive proofs . . . 51
4.3 Initial results . . . 53
4.4 Analysing failure . . . 55
4.4.1 Deconstructing $\texttt{nextR}_\texttt{s}$ . . . 56
4.4.2 Well-formedness of programs . . . 60
4.5 Failure implies divergence . . . 61

4.5.1 Badly-formed programs . . . 62
4.5.2 Failure of actions . . . 63
4.5.3 Well-formed failure of `nextR`$_5$ . . . 63
4.6 Confluence of reduction . . . 68
4.7 Chapter summary . . . 71

5 A toolkit for building I/O models 73
5.1 Simple combinators . . . 74
5.1.1 Cartesian product combinator . . . 75
5.1.2 `String` map combinator . . . 75
5.1.3 Other possible combinators . . . 79
5.2 Attempting dynamic allocation . . . 80
5.2.1 The problem . . . 80
5.2.2 A basic solution . . . 81
5.2.3 Usage of `dmap` . . . 84
5.3 Location-based I/O models . . . 86
5.3.1 The approach . . . 86
5.3.2 Definition and relationship to normal I/O models . . . 88
5.3.3 Simple location-based combinators . . . 91
5.4 The `dmap'` combinator . . . 92
5.4.1 Pools . . . 92
5.4.2 Definition . . . 94
5.5 Chapter summary . . . 96

6 A real world API and applications 97
6.1 Dissecting Haskell's I/O API . . . 97
6.1.1 I/O primitives in Haskell 98 . . . 97
6.1.2 Communications primitives . . . 100
6.1.3 Extending the Haskell interface . . . 103
6.2 Modelling an individual file . . . 103
6.2.1 The file world state and I/O contexts . . . 104
6.2.2 File actions . . . 106
6.2.3 Confluence of file model . . . 107
6.3 Two communication primitives . . . 110
6.3.1 Communication channels . . . 110
6.3.2 Quantity semaphores . . . 111
6.4 A unified I/O library . . . 113
6.4.1 Semantics of Haskell's I/O actions . . . 114
6.4.2 Using handles to organise concurrency . . . 115
6.4.3 Other issues . . . 118

6.5 Applications . . . 118
6.5.1 A file encoder with user interface . . . 118
6.5.2 A background log file . . . 122
6.5.3 Concurrent file processing . . . 123
6.5.4 Quantity semaphore example . . . 124
6.6 Chapter summary . . . 125

7 Axiomatic semantics **127**
7.1 A big-step semantics . . . 128
7.1.1 Preliminaries . . . 128
7.1.2 World/program equivalence . . . 130
7.1.3 Deriving a big-step semantics . . . 133
7.2 Weak program equivalence . . . 137
7.2.1 Definition of weak program equivalence . . . 137
7.2.2 Laws of weak program equivalence . . . 141
7.3 Full program equivalence . . . 143
7.3.1 Full equivalence rules . . . 143
7.3.2 Is full equivalence substitutive? . . . 143
7.4 Equivalence proofs for I/O actions . . . 146
7.4.1 Some small proofs . . . 146
7.4.2 A larger proof . . . 147
7.4.3 Non-termination . . . 149
7.5 Large-scale I/O proofs . . . 149
7.5.1 Domain-based I/O models . . . 150
7.5.2 Hybrid program equivalence . . . 150
7.5.3 Manipulating concurrent programs . . . 151
7.6 Chapter summary . . . 152

8 A lattice-theoretic approach to I/O contexts **153**
8.1 An axiomatisation of I/O contexts . . . 154
8.1.1 Definition . . . 154
8.1.2 Relation to I/O models . . . 157
8.1.3 Properties of inversion . . . 159
8.2 Defining a maximal lattice . . . 160
8.2.1 Examples . . . 163
8.3 Mechanisms for splitting contexts . . . 166
8.3.1 Context Splitter A . . . 167
8.3.2 Context Splitter B . . . 169
8.3.3 Discussion . . . 171
8.4 Future directions and related work . . . 171

8.4.1 Similarities to other algebras . . . 172

9 Conclusions and future work **175**
9.1 Conclusions . . . 175
9.2 Future work . . . 175
9.2.1 More exotic types . . . 176
9.2.2 GUI systems . . . 176
9.2.3 A simpler, core calculus . . . 176
9.2.4 Exceptions . . . 177
9.2.5 Other ideas . . . 177

A Implementation details **179**
A.1 Curio implementation . . . 179
A.1.1 Reduction in Curio . . . 179
A.1.2 Re-implementing $\texttt{next}_{\mathfrak{s}}$ . . . 179
A.1.3 Re-implementing $\texttt{nextR}_{\mathfrak{s}}$ . . . 181
A.2 Combinators . . . 182
A.2.1 Map functions . . . 182
A.2.2 Pool functions . . . 183
A.2.3 Helper functions for $\mathsf{Cxt}\ \varsigma$ . . . 184
A.2.4 Determining a process' location . . . 184
A.3 Real world I/O semantics . . . 185
A.3.1 File action semantics . . . 185
A.3.2 I/O library semantics . . . 186
A.3.3 Implementation details for `parIO` . . . 188

B Additional proofs and machine-verification **191**
B.1 Additional maximal lattice results . . . 191
B.2 Full Abstraction and Admissibility . . . 195
B.2.1 Full abstraction . . . 195
B.2.2 Admissibility . . . 196
B.3 Sparkle proof sections . . . 197

References **197**

List of Figures

1.1 Definition of I/O model $\mathfrak{term}$ . . . 11
1.2 Chapter dependency diagram . . . 14

2.1 Relations on actions and contexts . . . 24
2.2 The pre-condition $\mathsf{PRE}_{\mathfrak{s}}$ . . . 25
2.3 $\mathfrak{bffr}$ – a 1-to-1 communication buffer . . . 25
2.4 $\mathfrak{lock}$ – a mutex . . . 26
2.5 $\mathfrak{ivar}$ – a shared integer variable . . . 27
2.6 $\mathfrak{istr}$ – an integer I-structure . . . 28
2.7 $\mathfrak{term}$ – a model for terminal I/O . . . 30

3.1 Non-deterministic single-step semantics for Curio . . . 38
3.2 Encoding details for Curio . . . 40

4.1 Definition of $\mathtt{next}'_{\mathfrak{s}}$ . . . 50
4.2 Annotating single-step reduction . . . 51
4.3 Definition of $\mathtt{nextR}'_{\mathfrak{s}}$ . . . 56
4.4 Well-formedness of programs . . . 60
4.5 Proving confluence using the diamond property . . . 70

5.1 The cartesian product of I/O models . . . 74
5.2 Sample cartesian product program . . . 76
5.3 Definition of `dmap` . . . 83
5.4 Helper functions for the `dmap` combinator . . . 85
5.5 Definition of location-based I/O models . . . 88
5.6 Converting between IOModel and IOModelL . . . 89
5.7 Definition of `dmap'` combinator . . . 94

6.1 Chosen Haskell 98 I/O actions . . . 101
6.2 High-level semantics of file actions . . . 108
6.3 $\mathfrak{chan}$ – a one-to-one communication channel . . . 112
6.4 $\mathfrak{qsem}$ – a quantity semaphore . . . 112

6.5 Implementation of unified I/O libraries (Part I) . . . 116
6.6 Implementation of unified I/O libraries (Part II) . . . 117
6.7 A file encoder in Curio (Part I) . . . 120
6.8 A file encoder in Curio (Part II) . . . 121
6.9 Definition of `foldlFile` . . . 124

7.1 Definition of world/program equivalence . . . 131
7.2 Equivalence of world/program pairs . . . 131
7.3 Big-step operational semantics for Curio . . . 134
7.4 Big-step operational semantics for Curio – divergence . . . 135
7.5 Convergence of parallel terms . . . 138
7.6 Derived full equivalence rules . . . 144
7.7 Additional pre-conditions . . . 151

8.1 I/O model "RSS_0" . . . 155
8.2 I/O model "RS_FS_T" . . . 156
8.3 The six maximal lattices of order 2 . . . 164
8.4 Two non-distributive maximal lattices . . . 165
8.5 Maximal lattices for example I/O models . . . 165
8.6 Maximal lattice for model "$ERSWZ$" . . . 166

A.1 Implementation of $\texttt{next}_\mathfrak{s}$ and $\texttt{rdce}_\mathfrak{s}$ . . . 180

B.1 Sparkle theorem names . . . 198

Acknowledgements

First and foremost, I would like to thank my supervisor Andrew Butterfield. He gave me a great research grant, a good sense of focus and direction, distributed plenty of useful advice, and never doubted the value and validity of my work.

I am in debt to the other members of the Foundations and Methods Group in Trinity College Dublin, and this dissertation could not have been written without the helpful feedback and sense of community which this small group provided. Cheers to Shane O'Conchuir for being an agreeable office-mate, and, in doing so, keeping me generally sane over the months and years. Glenn Strong was always good for random chats and for bouncing ideas off. Malcolm Tyrrell was very helpful during the early years, and I was sorry to see him go. Thanks to Colin Little for many an after-hours discussion about logic. My work has benefited also from discussions with Arthur Hughes, Hugh Gibbons and Robert Byrne. Thanks to Edsko for his XY-pic notes, and to Wendy for her proof-reading assistance.

My five month trip in 2003 to the University of Nijmegen, The Netherlands was extremely valuable. Warm thanks go out to Rinus Plasmeijer and Marko van Eekelen for making it possible, and Maarten de Mol for writing a nifty proof-assistant and answering my questions. The company of Artem, Mariusz, Luís, Lars, Iris, Sebastian and Marianne made the visit an enjoyable experience.

Slogans and encouragement to Tweek, Ste, Aidan Mc., Hamill, Michael, Mackers, Jenn, Colm, Aidan K, and Claire – the old CS crowd who I've been hanging around with quite a bit over the years, and with whom I've enjoyed many a good night out.

Finally, I wish to thank my family for their constant support and encouragement.

While working on this dissertation I was funded by Enterprise Ireland Basic Research Grant SC-2002-283.

"*I'll finish this jigsaw,*
I'll find the pieces behind the couch"
– from "White Water Song", Bell X1

Chapter 1

Introduction

Pure, lazy functional languages [90, 97] have often been praised for the elegant programming style which they encourage. These languages, through referential transparency and lazy evaluation, give one a means to write and manipulate programs almost at the level of a specification. Therefore one can write expressive code which is a clearer description of what we intend it to do, and, ideally, one only worries about optimisations when the program is logically correct.

But this cannot be said to hold for input/output. Monadic I/O [122], the standard approach to I/O in pure languages, forces the programmer to explicitly sequence all I/O actions. This is not abstract at all, and is hardly a better specification of I/O than any C or Pascal program. This opinion has been echoed by the functional language community. In Peyton Jones' 15 year retrospective on the Haskell programming language [87] he mentioned that a more fine-grained way of partitioning I/O effects was one of two significant open challenges associated with the use of monads.

The problem is that certain I/O actions genuinely do not interfere with one another and the programmer should be free to loosen the order in which these actions are performed. For example, let us say that we want to write a program which does file processing while, at the same time, interacting with the user. These are probably distinct, unrelated pieces of code, yet if actions are sequenced explicitly then either all user interaction will have to halt while the file is being processed, or both code fragments will have to be entangled together. Neither solution is particularly desirable. We also want to avoid adding unconstrained concurrency since this would lead to non-determinism.

This dissertation's contribution to this open problem is two-fold. Firstly, we outline a somewhat novel language extension which, using runtime checks, preserves determinism in the presence of concurrency and I/O. Secondly, and more significantly, this is achieved with what appears to be a genuinely original approach to describing the *semantics* of deterministic concurrency and I/O, thereby giving a rigorous foundation to our new language features. We begin by developing a structure called an "I/O model" which directly describes the API, and this is then used to give the semantics of a language called Curio. This language extends

a purely functional language by adding five primitives which together permit monadic I/O, concurrency, and interprocess communication yet still allows us to retain determinism. We then show how a significant subset of Haskell 98's I/O library may be modelled with this technique and how traditional notions of program equality and equational reasoning still hold true.

Section 1.1 gives a quick history of functional languages and how one expresses I/O in them. Section 1.2 describes the semantics of functional languages, and Section 1.3 presents a detailed overview of the differing attempts to give a formal semantics to I/O. Section 1.4 gives a broad outline of this dissertation, showing its main contributions and the contents of the individual chapters. Section 1.5 describes work related to that in this thesis. Section 1.6 contains technical preliminaries.

1.1 Functional programming and I/O

Functional programming is a language paradigm which treats computation as the evaluation of mathematical functions. Generally functional languages attempt to incorporate functions as "first class citizens" within the language, allowing them to be passed as parameters to other functions.

Early languages such as LISP [70] were strict (meaning that functions evaluated their arguments), had no static type-system, and included imperative features such as side-effecting I/O and assignment. ISWIM [64] introduced lazy evaluation, and ML [76] in the mid-70s was the first functional language with a polymorphic type system.

1.1.1 Pure functional languages and I/O

Gradually, over time, there has also been a shift towards removing or isolating imperative features. "Pure" languages are that class of functional language which place the greatest emphasis on the semantic properties of functional languages by entirely forbidding these useful but inelegant imperative features. In these languages it can be assumed that there is no implicit reduction order (see [101], for example), so I/O, assignment, and array manipulation need to be ordered explicitly using special language constructs.

Side-effecting I/O is the standard means of expressing I/O in almost all programming languages. With this approach I/O is performed by pseudo-functions, and these generally exist outside the realm of the language's semantics. For lazy functional languages, however, side-effecting I/O is out of the question. Consider the following lazy list, where `getString` is a side-effecting pseudo-function which retrieves a string from the terminal:

```
printlist :: [String]
printlist = ["foo", getString, "bar"]
```

Depending on how this list is used, `getString` may be called at once, at some time in

the future, or never. Therefore the programmer will most likely have little clue as to when, if ever, the I/O action is going to take place.

The 1980s saw a proliferation of pure functional languages but, after a period of consolidation in the academic community the Haskell language [90] has now become the focus of almost all research in this area. The one notable exception is the Clean language [97], which is similar but has evolved in different directions, especially with regard to state and I/O. Haskell and Clean both have polymorphic type systems, a type-class mechanism and (two different) facilities for performing I/O. This dissertation is only concerned with these pure languages.

We shall now describe the two main approaches to I/O, namely monads and unique types.

1.1.2 Monadic I/O

Monads are usually regarded as the single most significant breakthrough for the expression of I/O in pure lazy functional languages, and they are a fundamental starting point for this dissertation. Monads are a construct from category theory [68], and were originally introduced into the field of Computer Science by Moggi [77] as a proposal for structuring the denotational semantics of programming languages. Wadler and Peyton Jones later successfully applied them to functional programming [121, 94], where, among other uses, they let one explicitly sequence I/O actions. Monads are still the primary means by which I/O is expressed in Haskell.

A monad, in a functional language, is a type constructor with two operations defined on it, `return` and `>>=`. In Haskell one sequences all I/O actions using the `IO` monad, and all I/O must be conducted using built-in functions of type `IO` α.

$$\begin{aligned}
\texttt{return} &:: \forall\alpha.\ \alpha \to \texttt{IO}\ \alpha \\
\texttt{(>>=)} &:: \forall\alpha.\ \forall\beta.\ \texttt{IO}\ \alpha \to (\alpha \to \texttt{IO}\ \beta) \to \texttt{IO}\ \beta \\
\texttt{putStr} &:: \texttt{String} \to \texttt{IO ()} \\
\texttt{getChar} &:: \texttt{IO Char}
\end{aligned}$$

A term of type `IO` α can be understood as an I/O performing program which, upon completion, returns a value of type α. `return` a is a program which immediately returns value a, the built-in `IO` "programs" such as `getChar` each perform a single system call or side-effect, and one uses `>>=` to make a program which performs one program and then, depending on the result, perform another. So it is in effect sequencing by construction. This forms an entirely sequential outer shell around the inner core, functional language.

A good introduction to monadic I/O can be found in [122], although the Haskell syntax is slightly out of date.

1.1.3 Unique types

The other approach to I/O in pure languages is through unique typing [10, 5] and this is what the Clean community uses to structure I/O and array access. Unique types, derived from linear types [118, 37], add additional type information to terms which indicates whether access to a term must be single threaded.

Uniqueness is indicated with a `*` and the initial program `Start` is of the following type

$$\texttt{Start} :: \texttt{*World} \rightarrow \texttt{*World}$$

The great significance of this approach for our research is that one may loosen the sequence in which actions are performed. The types of some of the file actions are as follows*:

```
 fopen  ::  [Char] → Mode → *World → (Bool,*File,*World)
fclose  ::  *File → *World → (Bool,*World)
freadc  ::  *File → (Bool,Char,*File)
fwritec ::  Char → *File → *File
```

When a file is opened it becomes its own unique `*File` object, separate from the world of type `*World`. Therefore one may write a program which sequences the order in which actions are to be performed only with respect to an individual file. This is sometimes known as a world-as-value style. The oft-cited disadvantage, however, is that unlike monads it requires non-trivial extensions to the standard Hindley-Milner [25] type system.

1.1.4 Other approaches

Prior to monads, a collection of different styles of pure I/O were proposed. These included Landin stream I/O, synchronised stream I/O, and continuation-passing. Good introductions may be found in [94, 39], and they are formally proved equivalent in [38], but these are of little concern to us since they are no more powerful than monadic I/O.

Haskell permits lazy file I/O. The function `hGetContents :: FilePath → IO String` reads the contents of a file. However it returns immediately and the characters are only actually read from disc when the resultant string is evaluated. While convenient, this comes at a price – the behaviour of lazy file I/O, like with side-effecting, can be hard to predict.

There are also high-level libraries which are used to structure I/O. The best example of this is systems for handling graphical user interfaces, and these may have their own high-level semantics. Yet it appears that none of these systems *do* yield a more fine-grained approach to general I/O than the low-level language constructs used to implement them. Fudgets [15, 16] are perhaps the most famous GUI library. In Fudgets there are special terminal and

*The actual Clean types (and syntax) are a little different, containing extra strictness annotation.

file I/O fudgets to which there is unrestricted access – but Fudgets are non-deterministic so this is no different to normal concurrency. Arrows [54] are a generalisation of monads but actions are still sequenced explicitly, and GUI systems based on Arrows [20] are no different. The Clean language implements GUIs via the Object I/O library [6], and using the unique type system processes may contain, as part of their local state, the file system or individual files.

Also, despite the considerable work on composing monads [59] – inspired by category-theory – it has not helped the structuring of independent states within a single I/O monad.

1.2 Semantics of functional languages

1.2.1 Introduction to language semantics

The purpose of a language semantics is, in some sense, to describe formally how programs behave. This may then be used to give formal proofs of properties, show that a type system is sound, or show that compiler optimisations are valid. General-purpose introductions to the field of language semantics can be found in [43, 124]. Programming language semantics are generally classified as operational, denotational or axiomatic, and we will briefly discuss the first two here.

An operational semantics describes how a program behaves over time. Typically, an operational semantics defines a term structure (i.e. the syntax of a language), some reduction relation "$\longrightarrow$" on that structure, and some notion of a term being in normal form/fully evaluated. $p \Downarrow v$ means that term p reduces to normal form v (convergence), and $p \Uparrow$ indicates that p does not reduce to a normal form (divergence). Terms often have some associated type information as well.

A denotational semantics describes a program as an element of a mathematical structure. So if an operational semantics says what a program *does*, a denotational semantics could be said to describe what a program *is*. Very often this structure is a Scott-domain – a partially ordered set with some extra side-conditions, and a $\perp$ (bottom) element which denotes non-termination/failure. The denotation of a program p is written as $[\![p]\!]$. For this dissertation we will be concerned solely with traditional Scott domains. An introduction to denotational semantics can be found in Schmidt [102], and Abramsky and Jung's notes on Domain Theory [4] are probably the best reference text on the subject.

Various relationships may be proved which connect an operational semantics to a denotational one. Taking the terminology from Pitts' notes [95], "soundness" means that $p \Downarrow v$ implies $[\![p]\!] = [\![v]\!]$, "adequacy" means that $[\![p]\!] = [\![v]\!]$ implies $p \Downarrow v$ (where v is in normal form), and "full abstraction" implies that two programs are contextually equivalent under the operational semantics if and only if their domains are equal. Contextual equivalence is an operational notion. Two programs are contextually equivalent if substituting one for the other within any other program does not affect the observable results.

1.2.2 PCF and applicative bisimulation

The semantics of functional languages are sometimes said to be derived from the λ-calculus [9], Church's minimal notion of computable function. Yet in practice this correspondence is somewhat sketchy. The pure λ-calculus has no delta rules (atomic units of computation, such as manipulating the natural numbers), does not specify any exact reduction strategy, and has no type system.

PCF, introduced by Plotkin in [98], includes all of these and is instead usually viewed as the best example of a minimal functional language. Plotkin gave an operational and denotational semantics to PCF using Scott-domains, and proved soundness and adequacy results linking the two semantics. Famously, however, full abstraction did not hold for PCF under standard domain theory[†]. Denotational equality does still imply contextual equivalence but the reverse is not true.

In lazy functional languages one evaluates a term only to its outermost constructor. Abramsky's Lazy Lambda Calculus [1] re-addressed the problem of treating the λ-calculus as a real language by developing a theory in which reduction is only to weak head normal form – that is, until an outermost λ. In doing so he developed the notion of "applicative bisimulation", which views the behaviour of a term as transitions in a process calculus like CCS [73]. This describes two terms as being equivalent if either (1) they both diverge, or (2) they both converge to the same outermost constructor and all sub-terms are equivalent in that same sense.

This general approach was explored in an entirely operational setting for a class of lazy computation systems by Howe [51], who showed the circumstances in which bisimulation is a congruence and when it coincides with contextual equivalence. This technique was used by Gordon [39, 40] to describe and give good notions of program equivalence to a substantial lazy functional language.

There also exist more low-level semantics. Launchbury's Natural Semantics [65] describes the heap usage of lazy evaluation, and [105, 63, 92] are abstract machines for lazy evaluation. These are not relevant to our work.

1.2.3 Proving program properties

These minimal languages are ideal for meta-results and proof-of-concept tasks, but formal semantics for full languages are harder to come by. The Haskell Report [90] describes the Haskell language semi-formally but there is still, as of yet, no complete formal semantics for Haskell. This is ongoing work, and some significant documents on this subject are [44, 93, 33, 47]. The semantics of Core-Clean, a language with a large subset of the functionality of Clean, can be found in de Mol's (currently unfinished) dissertation [27].

[†]The problem of giving a fully abstract semantics to PCF has only been solved quite recently [3], and a survey of the various attempts can be found in [84].

When one proves general properties via a language's operational semantics it is commonly by inducting over the term structure of the language, or the number of reduction steps. In a denotational style, one instead inducts over the structure of the domain of a universally quantified variable.

Equational reasoning in pure functional languages is usually performed in this denotational style. This is the type of reasoning popularised, for example, in Bird's introductory textbook [12], where one proves that

$$\forall_{f \in \alpha \to \beta}.\forall_{xs \in [\alpha]}.\forall_{ys \in [\alpha]}.\texttt{map}\ f\ (xs +\!\!+ ys) = \texttt{map}\ f\ xs +\!\!+ \texttt{map}\ f\ ys$$

by inducting over xs. xs may be any element of the domain $[\alpha]$, the domain of all (possibly infinite, possibly partial) lists of α. Valid lists include $[1,2,3]$, $[\bot]$, $[7,7,7,\cdots]$ and $(\bot{:}\bot)$.

Equational proofs need not always rely on domain theory. Gordon, in [40], proves the so-called "Bird & Wadler Take-lemma" via operational means using bisimulation.

As far as we are aware there are only two tools in existence for reasoning about pure, lazy functional programs. These are

- Sparkle [28], which is a stand-alone application for reasoning about programs in CoreClean, a language with a large subset of the functionality of Clean.
- The Programatica [85] tool which is used to let one reason about Haskell using P-Logic. This must then be translated into a separate type theory before reasoning may be performed.

It is the former, Sparkle, which is used to help verify the results in this document.

1.3 Semantics for I/O

I/O is not generally considered to be an interesting aspect of language semantics. The need for a semantics for I/O is more acute with pure functional languages, however, and there are a number of approaches.

1.3.1 I/O as state manipulation

The simplest and most obvious way of modelling I/O is as a state-transformer, where some "world state" ω models the entire system state, and the semantics of each action is a function of type $\omega \to (\omega, \nu)$. So, given some system state an action modifies it and returns some value of type ν. One of the cited disadvantages with a state-transformer semantics is that it is not compatible with concurrency, interaction and non-termination – but this dissertation will contest some of these.

There are a number of elegant techniques for expressing state manipulation in functional languages which also happen to be applicable to I/O. This is true of Clean's unique types.

Haskell's lazy state threads [66], a well-known technique for dealing with state, were modified in [50] to allow concurrent, single threaded access to *global* sub-states, such as individual files. Composable Memory Transactions [46] use a form of memory transaction to permit concurrent data-structures, but probably are not applicable to everyday I/O.

General-purpose state-based techniques are inadequate since, as a semantics for I/O, they ignore the fact that I/O is fundamentally different to state: memory modification can be rolled back (unlike, say, deleting a file); one can change memory whatever way one wants, but global state can only be changed via a fixed interface; I/O is inherently global but state may be allocated, modified and deallocated entirely locally. That does not mean that these techniques cannot be used in an ad-hoc manner to solve certain problems elegantly. It just means that they do not describe the complexity of I/O very accurately.

1.3.2 Bisimulation – the Haskell approach

Probably the single most significant attempt at a semantics for functional I/O is to be found in Gordon's Ph.D thesis "Functional Programming and Input/Output" [39]. In this document he develops a full operational semantics for a pure functional language using bisimulation and extends this notion of bisimulation so that transitions, or observations, may include I/O actions. In this system, attributed originally to Holström [49], all actions are observable events and two different sequences of actions are always distinguishable. A denotational semantics was given for this system in [23].

A similar approach was used to give the semantics to Concurrent Haskell [89], and, later, describe the details of Haskell's I/O, exception and foreign interface mechanisms [88]. This is similar in the sense that I/O actions are transitions. The difference is that it uses an implicit *denotational* semantics of the host language (i.e. Haskell), instead of giving a full operational semantics.

This gives a satisfactory model of user interaction and non-termination. The main flaw to this semantics is its usefulness since it "explicitly represents the instructions issued by a program, rather than their observable effect" [89]. This style of semantics was used to prove the monad laws in [40]. However, even in Gordon's Ph.D. thesis, in order to prove useful properties about the *effects* of I/O actions a state-transformer semantics had to be used.

1.3.3 Other approaches

There have been occasional attempts at reasoning about side-effecting I/O. Williams and Wimmers [123] give a semantics to side-effecting I/O in the language FL, a modification of Backus' FP. However, since FL is a strict language this is much simpler. Schmidt-Schauß [103] attempts to gives a semantics to lazy side-effecting (such as `unsafePerformIO` in Haskell) but it is inherently non-deterministic.

Sewell [106] addresses I/O in Pict, a language based on Milner's π-calculus [75]. He develops a highly simplified notion of a sequential C program and the UNIX X-Windows

request buffer, and proves an abstract machine correct with respect to the semantics. This is mostly the same as the bisimulation approach, as is that taken in [116], where the λ-calculus is extended with simple I/O primitives.

There also exist techniques for reasoning about lazy stream I/O. Thompson [115] gives a trace-based semantics to lazy stream I/O in Miranda. Hall and Hammond's draft dynamic semantics for Haskell 1.3 [44] describes the effect of I/O actions with respect to a single system state, and this includes a semantics for file system actions. Hudak also does something similar in [52]. These aforementioned semantics for Haskell describe specifics of the I/O API but none use this to exploit concurrency.

Some related work is that on functional operating systems, since this in effect gives the semantics to part of the operating system by implementing it in a functional style. In the 1980s there was a considerable amount of work in this field [62, 61, 24]. Recently, pure, strongly-typed functional operating systems have been developed such as Famke [120] and House [45]. These require a more detailed examination of the features of the I/O interface, but their semantics do not appear to be compatible with describing deterministic concurrency.

Some work has gone into giving formal semantics to GUI systems. Semantic models of Fudgets were given in [78] and [111] but these are non-deterministic and do not describe the actual API.

Languages such as Vault [29] and Cyclone [42], though unrelated to pure functional languages, are also notable since they have a strong notion of state. Vault is a modification of C and has been used to provide a more secure interface to Windows 2000 device drivers. Although their type systems are sensitive to specific aspects of the I/O API, they do not go quite so far as to give a semantics to I/O.

1.4 Thesis outline

1.4.1 A brief technical introduction to Curio

In this document we outline a small language extension, and its associated semantics. The Curio language defined has five primitives, `>>=`, `return`, `action`, `test` and `par`. Their types are

$$
\begin{aligned}
\texttt{(>>=)} &:: \forall\beta.\, \forall\gamma.\, \mathsf{Prog}_{\mathfrak{s}}\ \beta \to (\beta \to \mathsf{Prog}_{\mathfrak{s}}\ \gamma) \to \mathsf{Prog}_{\mathfrak{s}}\ \gamma \\
\texttt{return} &:: \forall\beta.\, \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \\
\texttt{action} &:: \alpha \to \mathsf{Prog}_{\mathfrak{s}}\ \nu \\
\texttt{test} &:: \forall\beta.\, \alpha \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \\
\texttt{par} &:: \forall\beta.\, \forall\gamma.\, \forall\epsilon.\, \rho \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \gamma \to (\beta \to \gamma \to \epsilon) \to \mathsf{Prog}_{\mathfrak{s}}\ \epsilon
\end{aligned}
$$

Like '`IO`' in Haskell, $\mathsf{Prog}_{\mathfrak{s}}$ is a monadic type constructor. $\mathfrak{s}$ identifies the Curio pro-

gram's "I/O model" and this describes the I/O interface and binds the types ν, α and ρ. Primitives `>>=` and `return` have the same roles as in Haskell, and `action` a performs primitive action a. We achieve determinism in Curio by limiting the actions that processes can perform, and the `test` primitive lets a process check at run-time if a certain action is allowed. Lastly, `par` performs two programs concurrently, giving each sub-process certain permissions.

As a small example, consider the following program for I/O model $\mathfrak{term}$ which attempts to write a string to `stdout`, returning `False` if that is not allowed.

```
putStr' :: String → Prog_term Bool
putStr' []     = return True
putStr' (c:cs) =
  test (PutC c) (action (PutC c) >>= \_ -> putStr' cs) (return False)
```

The types α, ν and ρ, the behaviour of all the actions, and the amount of deterministic concurrency allowed are determined by the following abstract I/O model structure.

$$\begin{aligned}\mathfrak{s} :: \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \quad \triangleq \quad & \langle \mathtt{af} :: \alpha \to \omega \to (\omega, \nu), \\ & \mathtt{wa} :: \alpha \to \omega \to \mathsf{Bool}, \\ & \mathtt{ap} :: \varsigma \to \alpha \to \mathsf{Bool}, \\ & \mathtt{pf} :: \rho \to \varsigma \to (\varsigma, \varsigma) \rangle\end{aligned}$$

In short, `af`/`wa` describe the effect of each action on global state (including when an action is stalled), `ap` describes what permissions we can give to individual processes, and `pf` describes how these permissions can be distributed in the presence of concurrency. The intricacies of I/O models and Curio's primitives are examined in great detail in Chapters 2 and 3, and this forms the basis for the whole of the rest of the dissertation.

The somewhat involved semantics of model $\mathfrak{term}$ can be found in full in Figure 1.1. Each character sent to `stdout` gives rise instantly to zero or more characters, and these are added to a queue of characters ready to be consumed from `stdin`. A process may or may not be able to output characters, and, separately, may or may not be able to input characters. The I/O model guarantees that two parallel processes cannot both be able to perform input, or both be able to perform output.

Model $\mathfrak{term}$ only has two actions, and very limited opportunities for concurrency. In Chapter 6 we construct a model $\mathfrak{io}$ which encodes Haskell's I/O interface. $\mathsf{Prog}_{\mathfrak{io}}$ can then be seen as a substitute for Haskell's `IO`, and we implement the standard Haskell functions such as `getChar` using Curio's built-in `action` primitive. Furthermore, we introduce some Curio-specific library functions for deterministic concurrency and communication, and these give

$$\mathfrak{term} :: \mathsf{IOModel}\ \nu_{\mathtt{Term}}\ \alpha_{\mathtt{Term}}\ \rho_{\mathtt{Term}}\ \omega_{\mathtt{Term}}\ \varsigma_{\mathtt{Term}} \triangleq \langle \mathtt{af}_{\mathtt{Term}}, \mathtt{wa}_{\mathtt{Term}}, \mathtt{ap}_{\mathtt{Term}}, \mathtt{pf}_{\mathtt{Term}} \rangle$$

$$\begin{array}{rcl@{\qquad}rcl} \nu_{\mathtt{Term}} & \triangleq & \mathsf{Char} & \alpha_{\mathtt{Term}} & \triangleq & \textsc{PutC}\ \mathsf{Char} \mid \textsc{GetC} \\ \omega_{\mathtt{Term}} & \triangleq & (\mathsf{[Char]}, \mathsf{TermIO}) & \rho_{\mathtt{Term}}, \varsigma_{\mathtt{Term}} & \triangleq & \textsc{TCxt}\ \mathsf{Bool}\ \mathsf{Bool} \end{array}$$

$$\mathsf{TermIO} \triangleq \textsc{TermIO}\ (\mathsf{Char} \to (\mathsf{[Char]}, \mathsf{TermIO}))$$

$$\begin{array}{lllcl} \mathtt{wa}_{\mathtt{Term}} & (\textsc{PutC}\ c) & (cs, t) & \triangleq & \textsc{False} \\ \mathtt{af}_{\mathtt{Term}} & (\textsc{PutC}\ c) & (cs, \textsc{TermIO}\ f) & \triangleq & ((cs +\!\!+ \mathtt{fst}\ (f\ c), \mathtt{snd}\ (f\ c)), \text{' '}) \\ \mathtt{wa}_{\mathtt{Term}} & \textsc{GetC} & (cs, t) & \triangleq & \mathtt{null}\ cs \\ \mathtt{af}_{\mathtt{Term}} & \textsc{GetC} & ((c : cs), t) & \triangleq & ((cs, t), c) \\ \\ \mathtt{ap}_{\mathtt{Term}} & (\textsc{TCxt}\ b\ \_) & (\textsc{PutC}\ c) & \triangleq & b \\ \mathtt{ap}_{\mathtt{Term}} & (\textsc{TCxt}\ \_\ b) & \textsc{GetC} & \triangleq & b \end{array}$$

$$\begin{array}{l} \mathtt{pf}_{\mathtt{Term}}\ (\textsc{TCxt}\ bp_p\ bg_p)\ (\textsc{TCxt}\ bp\ bg) \triangleq \\ \quad (\textsc{TCxt}\ (bp\ \mathtt{\&\&}\ bp_p)\ (bg\ \mathtt{\&\&}\ bg_p), \textsc{TCxt}\ (bp\ \mathtt{\&\&}\ \mathtt{not}\ bp_p)\ (bg\ \mathtt{\&\&}\ \mathtt{not}\ bg_p)) \end{array}$$

Figure 1.1: Definition of I/O model $\mathfrak{term}$

a more user-friendly interface to `action`, `test` and `par`. The new functions include:

$$\begin{array}{rcl} \mathtt{newChannel} & :: & \mathsf{Prog_{io}}\ (\mathsf{Handle}, \mathsf{Handle}) \\ \mathtt{newQSem} & :: & \mathsf{Prog_{io}}\ (\mathsf{Handle}, \mathsf{Handle}) \\ \mathtt{hAllowed} & :: & \mathsf{Handle} \to \mathsf{Prog_{io}}\ \mathsf{Bool} \\ \mathtt{parIO} & :: & \forall\beta.\ \forall\gamma.\ [\mathsf{Handle}] \to \mathsf{Prog_{io}}\ \beta \to [\mathsf{Handle}] \to \mathsf{Prog_{io}}\ \gamma \to \mathsf{Prog_{io}}\ (\beta, \gamma) \end{array}$$

These allow a process to create one-to-one communication buffers (with separate read and write handles), check at runtime if it may use a handle, and split into two separate deterministic concurrent processes each with their own permissions.

1.4.2 Hypothesis

This dissertation is an attempt to show the following hypotheses:

- that it is possible to formally describe a pure functional language with I/O and concurrency using a general mathematical model of the API.
- that a slightly modified state-transformer semantics is suitable for modelling both the effects of I/O actions and interprocess communication, and with suitable language constructs this may then be used to enforce determinism.
- this can be be applied to model and enforce determinism in a substantial I/O API, not just small academic examples.

- useful notions of program equivalence exist for such a language.

1.4.3 Synopsis

The document is structured as follows.

Chapter 2: I/O models and contexts We introduce our "I/O model" mathematical structure in this chapter. This models the API with which a program interacts and also the opportunities for deterministic concurrency that the API exhibits. A pre-condition PRE_s is described which, we argue, guarantees determinism if it holds for an I/O model. Five small example models are then given.

Chapter 3: Curio – a language for reasoning about I/O This chapter uses the notion of an I/O model to give the non-deterministic single-step semantics of a small language called Curio. This extends a pure functional language with five extra primitives: `return`, `>>=`, `action`, `test` and `par`. We give some example programs, encoding details, and a proof of recursive enumerability.

Chapter 4: Confluence We give a full machine-verified proof that PRE_s guarantees confluence in Curio.

Chapter 5: A toolkit for building I/O models After creating small confluent models, often it is convenient to construct larger I/O models directly from these. We describe a collection of "combinators" which do this, all of which are proved to preserve confluence. The problems with dynamic allocation in a single global state is addressed, and we develop the notion of a "location-based I/O model" which solves these issues by allowing actions to distinguish different calling processes.

Chapter 6: A real world API and applications The purpose of this chapter is to encode a substantial subset of Haskell 98's I/O library into Curio. This includes terminal I/O, a file system and two types of deterministic communication primitive – one-to-one channels and many-to-one quantity semaphores. Combinators are used to combine these into a single API, and we give a more high-level interface which mimics that of Haskell. We finish with four real world applications which demonstrate the power of Curio.

Chapter 7: Axiomatic semantics This chapter examines Curio's semantics from the point of view of reasoning about programs. We first develop a big-step semantics which relates the evaluation of a program to the evaluation of its sub-terms. Then we develop some notions of program equivalence which we find must be of a co-inductive nature since a program may communicate. We show how the monad laws hold under this equivalence relation and how it is a congruence. The chapter concludes with some example proofs for small I/O models and discusses future large-scale proofs.

Chapter 8: A lattice-theoretic approach to I/O contexts We re-examine our somewhat crude notion of I/O context (i.e. the permissions given to each process) from a more algebraic viewpoint. A structure called a "maximal lattice" is developed which neatly describes the various forms of concurrency permitted by an I/O model, and two general-purpose mechanisms are developed for giving permissions to sub-programs. We give examples and make comparisons with existing algebras.

Chapter 9: Conclusions and future work Conclusions are drawn and further work is suggested.

Appendix A: Implementation details Omitted definitions from Chapters 3-6 are given in full.

Appendix B: Additional proofs and machine-verification We give some extra results from Chapter 8, describe some formalities related to reasoning about functional languages, and show how the machine-readable forms of many of the proofs in this document may be obtained.

1.4.4 Results

This dissertation's main contribution is a full semantics for a lazy functional language with I/O and deterministic concurrency which admits correctness proofs. The original results contained in this dissertation are as follows:

- the semantics for a non-strict language with monadic I/O, concurrency and interprocess communication which is defined using an abstract mathematical model of the application programmer interface.
- the proof of confluence that with added runtime checks which obey particular axiomatic properties, the language will remain deterministic in the presence of concurrency.
- a comprehensive description of how the various aspects of a "real world" API may be modelled to give more flexible language design without non-determinism, allowing the user to write more expressive programs.
- the use of confluence to develop useful notions of program equivalence so as to admit correctness proofs for I/O which concern the actual API.
- Machine-verification of a significant number of technical results.

1.4.5 How to read this dissertation

Figure 1.2 shows the chapter dependencies. A normal arrow indicates a strong dependency between chapters, and a dotted arrow indicates a very slight dependency. A good

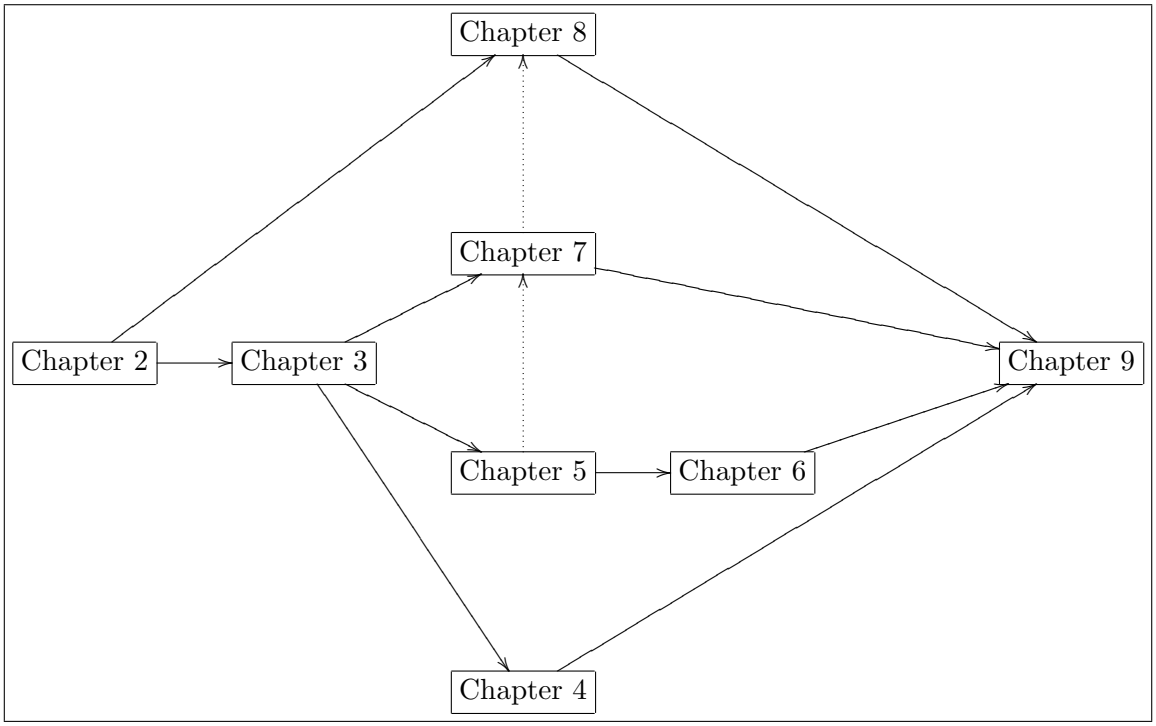


Figure 1.2: Chapter dependency diagram

understanding of Chapters 2 and 3 are mostly essential for the whole of the dissertation. After that, however, this document splits into a number of essentially distinct components. Chapters 5 and 6 are focussed towards constructing a practical, real world model of I/O. Chapters 7 and 8 are unrelated to these, and are concerned with formal reasoning and high-level algebraic properties of programs and I/O models. The confluence proof in Chapter 4 is largely self-contained.

1.4.6 Separate publications

A paper "Reasoning about Deterministic Concurrent Functional I/O" [30] outlining a much simpler prototype of Curio was accepted for the final proceedings of IFL 2004. This language did not permit any form of communication, and we did not discuss large-scale I/O models or mention notions of program equivalence. The confluence proof in Chapter 4 is considerably more complex than that found in [30] as a result of communication. We intend to publish this proof at a later stage, along with this dissertation's other original results.

1.5 Related work

Curio's main contributions are in the field of deterministic concurrency and the problems with giving a semantics to/reasoning about I/O (or global state), especially in the presence of concurrency. We now compare and contrast the work in this document to existing research

in the field.

1.5.1 Deterministic concurrency

Attempting to structure side-effects in a confluent way is an old problem and there are many existing approaches.

Clean is, for me, by far the most important and inspiring contribution towards expressing functional I/O concurrently. But, unlike Curio, there is no way for two separate sub-programs performing I/O on a separate piece of world state to communicate with one another – all communication must be performed at a synchronisation point at which the two sub-programs become one. Furthermore, Clean does not attempt to give a semantics to I/O as such. The I/O interface is constructed in an ad-hoc style based on Clean's graph-rewriting semantics [96].

The Brisk Haskell Compiler uses rank-2 polymorphism to guarantee deterministic access to multiple global sub-states [50, 109, 17]. This permits deterministic access to individual files. However, the language description is largely implementation-driven, with no semantics or notion of program equivalence. Eleni Spiliopoulou describes in her Ph.D. thesis [109] how concurrent threads may communicate deterministically, but this is just an implementation, and there are no associated semantics, nor are there any example programs. Also, shared file reads are just performed using lazy file I/O with only a brief, informal argument justifying why this is deterministic. Although lazy file I/O is not mentioned in Curio, we still describe shared file reads in a far more rigorous fashion.

Dataflow languages, such as Lucid and Id, are generally deterministic and functional in nature, and dealing with state has been found to be somewhat problematic. One of the most important contributions in this field, according to the history and a survey in [58], was the notion of an I-structure [8] introduced in the Id [81] language. These are single assignment structures which allow for the concurrent, yet deterministic construction of arrays. We show in Chapter 2 how I-structures may be modelled without difficulty in Curio's more general framework.

In a recent paper, Terauchi and Aiken [112] give a means of structuring side-effects under lazy evaluation via "witnesses". An ordering is enforced on side-effects by making one "see" a witness of the other. However, confluence is only statically decidable for some reduction strategies (call-by-need, but not call-by-name, for example), and it has only been applied to simple read and write cells, not full I/O. It is also unclear how easy this is to reason about formally. Programs in Curio are always confluent regardless of reduction strategy, and can cater for a variety of I/O mechanisms. But this still looks promising.

Type and effect systems are a standard approach to merging imperative and functional features, where pure and side-effecting terms are distinguished by the type system. Since the original work by Gifford and Lucassen in the mid-1980s [36, 67] much work has been done in the field [117, 22, 108]. Yet all of these rely on an *implicit* program evaluation strategy

and thus are not directly applicable to pure, lazy functional languages. An introduction to type and effect systems can be found in [79].

Segura [104] showed the correctness of determinism analyses in Eden, a pure (but non-deterministic) functional language with explicit parallelism. Since Eden does not permit I/O, however, this is not directly relevant. Jones and Hudak [60] discuss the issues regarding monadic I/O and explicit parallelism in pure functional languages – in particular, single threading problems. Concurrent Haskell adopted an approach quite similar to that mentioned – namely, it did not attempt to constrain concurrent behaviour. Concurrent Haskell is non-deterministic, but it is interesting nonetheless to read the justification for this decision.

> "Since the parent and child processes may both mutate (parts of) the same shared state (namely, the world), `forkIO` immediately introduces non-determinism... Whilst this non-determinism is not desirable, it is not avoidable; indeed, every concurrent language is non-deterministic. The only way to enforce determinism would be by somehow constraining the two processes to work on separate parts of the state ... The trouble is that *essentially all the interesting applications of concurrency involve the deliberate and controlled mutation of shared state*, such as screen real estate, the file system, or the internal data-structures of the program." [89]

Although it is true that a great many applications are non-deterministic by definition, state may still be shared in safe, deterministic ways, and we believe it is worthwhile to examine these in detail.

1.5.2 Reasoning about I/O and/or concurrency

Curio's desire to allow formal proofs about I/O (or global state) in the presence of concurrency is its other salient feature.

The only examples of large-scale reasoning about I/O in functional languages come from Trinity College Dublin. Butterfield and Strong [14] performed a case study comparing the ease of formal reasoning about I/O in C, Haskell and Clean. The author followed this with a larger proof of a simplified version of the UNIX `make` utility [32], which involved the development of some "reasoning operators" for sequential monadic I/O[‡]. Also, some very small machine-verified proofs of sequential monadic I/O programs can be found in [31].

The monad laws were proved by Gordon in [40], and in [39] he gave some small proofs describing the behaviour of a monadic controller used in an Edinburgh hospital. The only other example the author could find concerning formal I/O proofs are associated with the House functional operating system [45]. Small propositions concerning sequential monadic code can be proved correct using Programatica.

[‡]I was of two minds about whether to include details of this paper in an appendix to this dissertation. In the end I decided against it since it is not (yet) directly relevant to Curio.

The above examples all relate to purely sequential I/O, but Curio allows concurrency, and reasoning about concurrency is notoriously hard. Standard "pure" process calculi such as CCS [73, 74], CSP [48] and the π-calculus [75] are traditionally non-deterministic, and program properties are proved via trace semantics and bisimulation. Although very interesting in their own right, the sort of reasoning these permit is limited because it ignores the inherently state-based nature of many I/O proofs. Curio's notion of program equivalence developed in Chapter 7 is powerful since it can distinguish two program fragments both by how they modify world state and by how they communicate with other processes.

Curio's use of a state-transformer semantics to handle communication is unusual. Our equivalence relation may turn out to be a useful hybrid of stateful, inductive notions of I/O such as file access and interactive, co-inductive notions of I/O such as communication. The performance of large-scale I/O proofs in Curio is still future work, however.

1.6 Technical preliminaries

1.6.1 General prerequisites

The reader is expected to be reasonably familiar with Haskell syntax, and standard constructs from functional languages. In particular: higher-order algebraic datatypes, curried functions, type constructors, data constructors, lambda abstraction, and case analysis.

Function application in Haskell is written as `f x`. This is the same as `f $ x`, explicitly using the application operator `($)` which has the lowest precedence. Function composition is `(.)`. As usual, function application is left associative and the function type constructor $\rightarrow$ is right associative. Function definitions are given in the standard style. A single function may be defined by pattern matching on its arguments. This is internally converted into a `case-of` statement. A function may fail if no pattern matches, and the usual top-to-bottom scanning rules always apply. Lambda abstraction is written using a `\` (i.e. `\n -> n+1`). A '`_`' indicates a function argument whose value is ignored. A standard function which requires two arguments may be written in an infix notation using backticks (`f ‘map‘ xs` instead of `map f xs`). Similarly, an infix operator may be used in a prefix style (`(+) 2 3` instead of `2 + 3`).

The empty list is denoted `[]`, and `(x:xs)` is a cons cell. `[1,2,3]` is shorthand for `(1:(2:(3:[])))`. `length` returns the length of a list, `!!` is list look-up at a particular index, and list concatenation is $+\!\!+$, or `++` in Haskell. `head` and `tail` take the head and tail of a cons list, and `last` returns the last element in a list. A String is just a list of characters. The standard `map`, `foldl`, and `foldr` functions are assumed. Boolean (Bool) operators in Haskell are `&&`, `||`, and `not`, and general equality is `==` and `/=`. Integer operations exist as normal. Tuples are written (a,b), and `fst`/`snd` are the respective projection functions. The type () is the type whose only element is the value `()` (and $\perp$, of course). The type Either α β is either Left a or Right b, where a and b are of type α and β respectively. A

term of type Maybe α is either NOTHING or JUST a, where a is of type α.

The monadic sequencing function `>>` is defined as `m1 >> m2 = m1 >>= \_ -> m2`. "Do notation" is a convenient way of expressing monadic program code. The program fragment `do {v1 <- m1; m2}` really means `m1 >>= \v1 -> m2`, and `do {m1; m2}` translates to `m1 >> m2`. This may also be written without `{`s and `}`s using Haskell's vertical layout style.

Knowledge of more advanced topics such as rank-n polymorphism, the type-class mechanism, existential types and generalised algebraic datatypes is not a necessity.

For smaller program fragments we adopt different fonts to make the presentation clearer. For larger programs we revert to verbatim font for the entirety. We use different fonts to distinguish

- Types and type constructors (Bool, Maybe Int).
- Data constructors (TRUE, JUST x).
- Type variables in the types of polymorphic functions (β, γ, δ).
- Variables (i.e. as arguments to function definitions or as bound by quantifiers in propositions) (xs, i).
- Functions and built-in primitives (`fst`, `length` xs).
- I/O models ($\mathfrak{bfft}$, $\mathfrak{file}$, $\mathfrak{io}$).
- Sparkle "macros" ($\mathsf{PRE}_{\mathfrak{s}}$, $\mathsf{ally}_{\mathfrak{s}}(a_1, a_2)$).

Throughout this document the symbol '$\triangleq$' always means "is defined to be". We sometimes define algebraic datatypes in a convenient 'anonymous' notation – so, for example, instead of defining SomeType $\triangleq$ CNST1 | CNST2 and writing Maybe SomeType, we could just write Maybe (CNST1 | CNST2). Sometimes, for clarity, we explicitly insert quantifiers into the type declarations of polymorphic functions. Mostly, however, we just assume quantification at the outermost level for any type variable, which is the norm in Haskell.

We also assume that the reader is familiar with basic mathematics such as sets, functions and relations, and classical logic connectives such as $\wedge$, $\vee$, $\Longrightarrow$, and $\neg$.

Various sorts of ordered structure/relations are used in this document. A pre-order is a relation that is reflexive and transitive. An equivalence relation is a symmetric pre-order. A partial order is an anti-symmetric pre-order. A complete partial order D is a partial order if every chain in D has a least upper bound in D.

Domains are complete partial orders which possess a least element, usually denoted '$\bot$'. For example, the domain Bool has three elements, TRUE and FALSE, which are not comparable, and $\bot$. The domain Int is similar, except there are elements 0, 1, -1, 2, -2, ... as well as $\bot$. Both of these domains are examples of flat domains. That is, domains in which $x \sqsubseteq y$ if and only if $x = \bot$. As is common, we confuse a program's syntax with

its denotational semantics. In particular, we interchange $\perp$ with the syntax of a program whose denotation is $\perp$.

A lattice is a partial order in which every pair of elements has a least upper bound and a greatest lower bound. A function f is monotonic under a partial order $\sqsubseteq$ if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. Given a monotonic function f on the elements of a lattice, the set of elements x such that $x = f(x)$ (the set of fixpoints) forms a sub-lattice of the existing one.

1.6.2 LCF proof assistants

The Sparkle [28] proof assistant is LCF-based, which means that one reasons about programs using Scott domains. Milner introduced Stanford LCF [71] in the early 1970s, and this later evolved into Edinburgh LCF [41], which added a metalanguage (ML), and Cambridge LCF [86] which added the logical connectives $\vee$, $\exists$ and $\iff$. Paulson's book [86] is the standard text on the subject.

Sparkle is one of very few dedicated LCF-style theorem provers in existence and this made it probably the most appropriate tool for the work in this dissertation. Nowadays LCF-style proof is usually encoded in other more general logical frameworks and proof assistants, such as Isabelle [83] or Coq [114]. In Sparkle one reasons about programs in Core-Clean. Core-Clean is a simplified version of Clean, and is essentially identical to Haskell for our purposes. We adopt Haskell syntax instead since it will be more familiar to the average reader.

One reasons about programs in an equational style. Given two terms t_1 and t_2, if their types can be unified then $t_1 = t_2$ is a valid proposition which may eventually be proved correct. This equational style is then embedded within a first-order classical logic, with the standard connectives $\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\Longleftrightarrow$, $\forall$ and $\exists$ and the constants 'True' and 'False'. One can quantify over terms of any Core-Clean type, including functions. One can also quantify over propositions (i.e. one can prove that $\forall_{p_1 \in \mathbb{B}}.\forall_{p_2 \in \mathbb{B}}.p_1 \wedge p_2 \Longrightarrow p_2 \wedge p_1$). The type of a proposition is $\mathbb{B}$, and this is distinct from the type of any term.

Proofs are performed using a sequent calculus for natural deduction. The internal state of the proof is represented by a sequent of the form $\Gamma \vdash A$, where Γ is an unordered sequence of propositions. Tactics include:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \qquad \frac{\Gamma \vdash C \qquad \Gamma, C \vdash A}{\Gamma \vdash A}$$

Proofs proceed in a backwards style, so these rules should be read from bottom to top. The sequent calculus used is largely the same as that described in Section 2.13 of [86].

There is an important distinction to be made between True/False and TRUE/FALSE. True and False are the boolean constant propositions of type $\mathbb{B}$, which are actually rarely used; TRUE and FALSE are two elements of the three-element domain Bool, which also contains $\perp$. A term of type Bool (such as 1==2) is, on its own, not a proposition. However, the following three are: 1==2 = TRUE, 1==2 = FALSE, 1==2 = $\perp$. As in Sparkle, we may

use the shorthand notation 1==2 when we really mean 1==2 = TRUE.

One feature sorely missed in Sparkle is the ability to write macros which construct a large proposition from a basic recipe. One can use a normal Core-Clean function to construct one term from another, but we cannot do this to construct a proof obligation which contains quantifiers and other logical constructs. Throughout the document we occasionally emulate a sort of typed macro, defining a "function" of type $(\beta \to \mathbb{B})$. ':' is used, instead of '::' to distinguish the type of a proposition/macro from that of a term.

We do not give a rigorous description of the metalanguage, its type system and its logic because it is largely similar to that of existing approaches and we will never be proving properties at such a fine level of detail in this document that individual tactics will have to be mentioned.

Chapter 2

I/O models and contexts

The primary aim of this dissertation is the development of a functional language which permits I/O and concurrency, yet is deterministic and has a rigorous semantics. We begin by outlining the "I/O model" structure which will be used throughout this document. The purpose of an I/O model is to describe the API which a program interacts with when doing I/O. In particular, I/O models attempt to describe both the precise effect of each individual action on a "world state" and the extent to which certain actions are and are not order independent. This structure will be then used to give the semantics of a real language in Chapter 3.

In the literature, a state-transformer semantics for I/O is often adequate at a high-level for simple, purely sequential I/O [94, 66]. However, this is generally considered incompatible with issues such as concurrency and communication. In [89], Peyton Jones argues that

> "In practice ... it is crucial that the side-effects the program specifies are performed *incrementally*, and not all at once when the program finishes. A state-transformer semantics for I/O is therefore, alas, unsatisfactory, and becomes untenable when concurrency is introduced."

Haskell's I/O semantics is instead based on CCS [74]. This does not have any facilities for explaining the behaviour of actions or expressing the notion that two actions are order independent. Furthermore, the document which introduced this style of semantics for I/O, the Ph.D. thesis of Andrew Gordon [39], also had to abandon it for a state-transformer style in a later chapter in order to prove the very non-interference properties of actions which we ourselves want.

So, how do we describe the behaviour of I/O actions in a style that can distinguish actions that are order-independent yet still model communication and concurrency primitives? In this chapter we show that communication/concurrency may co-exist happily with state-transformers if we simply allow individual actions to become stalled. If actions can become stalled they can then synchronise with and communicate with other concurrent actions. One upshot is that state-transformers, for us, only describe the meaning of *individual* actions,

not entire programs, as is usually the case. The problem of giving a semantics to arbitrary I/O-performing programs is tackled (and solved) separately in Chapter 7.

Section 2.1 defines our I/O model structure, explains each individual component in turn, and introduces a pre-condition $\mathsf{PRE}_{\mathfrak{s}}$ which we argue guarantees determinism. Section 2.2 gives five small example I/O models which will be used throughout this dissertation – a communication buffer, a mutex, a shared integer variable, an I-structure and a model of terminal I/O. The chapter concludes in Section 2.3 with a more in-depth discussion of the features and flexibility of I/O models.

2.1 Definition

The definition of an I/O model in the metalanguage is the following 4-tuple parameterised by five types:

$$\begin{aligned}\mathfrak{s} :: \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \quad \triangleq \quad & \langle \mathtt{af} :: \alpha \to \omega \to (\omega, \nu), \\ & \mathtt{wa} :: \alpha \to \omega \to \mathsf{Bool}, \\ & \mathtt{ap} :: \varsigma \to \alpha \to \mathsf{Bool}, \\ & \mathtt{pf} :: \rho \to \varsigma \to (\varsigma, \varsigma) \rangle\end{aligned}$$

The five types take the following roles, all of which will be explained in more detail later:

- ν: return values from actions.
- α: primitive, atomic actions.
- ρ: "parameters" to the splitting of contexts.
- ω: world state.
- ς: I/O contexts.

As we shall see, these components are enough to describe state-based I/O which permits both a deterministic `fork`-like concurrency primitive and communication. It is important to remember that the five types are domains with a $\bot$ element, and the four functions can be any arbitrary computable function whose results may be undefined $(\bot)^*$.

The functions `af`, `wa`, `ap` and `pf` are now explained in detail[†].

*This probably isn't ideal, especially for the world model, but it is not serious. Any set-based model can be turned into a domain-based one by (1) turning the sets into "flat" domains by adding a bottom element, (2) making the functions bottom-preserving, and (3) showing that the functions are indeed Turing-computable.

[†]Unless otherwise stated, when giving general properties in this dissertation we always assume the existence of some arbitrary, implicit I/O model called $\mathfrak{s}$ which binds the five types ν, α, ρ, ω and ς, and the four functions `af`, `wa`, `ap` and `pf`.

2.1.1 The API – af and wa

The function $\texttt{af} :: \alpha \to \omega \to (\omega, \nu)$ defines the state-transformer for each action. Elements of type α identify atomic I/O actions which can be performed and for any action $a :: \alpha$, $\texttt{af}\ a :: \omega \to (\omega, \nu)$ is the state-transformer for that action. This describes how the action changes the global state of type ω, and what return value of type ν it yields. Type ν is typically a sum-type capable of storing Ints, Bools, Chars or any other value that an action might need to return.

If communication is to be permitted then there must be occasions in which an action is waiting for something to occur and cannot proceed. The function $\texttt{wa} :: \alpha \to \omega \to \mathsf{Bool}$ indicates exactly this. An action a can only be performed in world w when $\texttt{wa}\ a\ w = \textsc{False}$. We say an action a is **stalled** (in world state w) if $\texttt{wa}\ a\ w = \textsc{True}$.

Figure 2.1 defines two relations on actions and the API.

- $a_1 \mid\mid\mid_{\mathfrak{s}} a_2$: for any two actions a_1 and a_2, if neither are stalled then the order in which they are executed is irrelevant – both with regard to their effect on world state and their return values. $\mid\mid\mid_{\mathfrak{s}}$ is symmetric (but not necessarily transitive or reflexive) and is defined using a more general macro "cmmt". $\mathsf{cmmt}(f_1, f_2, w)$ says that state-transformers f_1 and f_2 are order independent when manipulating world state w.

- $\mathsf{ally}_{\mathfrak{s}}(a_1, a_2)$: if action a_1 is not stalled then performing it cannot cause action a_2, if also not stalled, to then become stalled. The word "ally" hints at the fact that action a_1 will not obstruct or hinder action a_2 – they are in effect working with each other.

The $a_1 \mid\mid\mid_{\mathfrak{s}} a_2$ condition allows for two possibilities: either both actions succeed for both orderings or for both orderings one action fails. For example, if $a_1 \mid\mid\mid_{\mathfrak{s}} a_2$, $\texttt{af}\ a_1\ w = (w_1, v_1)$ and $\texttt{af}\ a_2\ w = (w_2, v_2)$ then it is still possible that $\texttt{af}\ a_2\ w_1 = \bot$ and $\texttt{af}\ a_1\ w_2 = \bot$.

2.1.2 I/O Contexts – ap

The functions af and wa define the API. Since our aim is to guarantee deterministic I/O in the presence of concurrency, what is now needed is some means of reining in the power of a concurrent (sub-)program by giving it only a limited set of actions which it is allowed to perform. In general, actions are not order independent.

We call these permission sets **I/O contexts**[‡]. I/O contexts are elements of the type ς and the function $\texttt{ap} :: \varsigma \to \alpha \to \mathsf{Bool}$ defines the actions permitted by any context. A context c can be thought of as the set of actions a such that $\texttt{ap}\ c\ a = \textsc{True}$. Each (sub-)program has a context associated with it at runtime. The context determines what actions the (sub-)program is allowed to perform, and if used in a controlled fashion it gives a mechanism for ensuring determinism.

[‡]The author is aware that the word "context" is already overused in the field of functional programming (for example, evaluation contexts and type contexts). In this document, except in the rare case of "contextual equivalence", the word "context" will refer exclusively to I/O contexts.

$$
\begin{array}{rcl}
\mathsf{cmmt} & : & (\omega \to (\omega, \nu)) \to (\omega \to (\omega, \nu)) \to \omega \to \mathbb{B} \\
\mathsf{cmmt}(f_1, f_2, w) & \triangleq & \forall_{w_2 \in \omega}.\forall_{v_1 \in \nu}.\forall_{v_2 \in \nu}. \\
 & & \quad (\exists_{w_1 \in \omega}.f_1\ w = (w_1, v_1) \wedge f_2\ w_1 = (w_2, v_2)) \\
 & & \quad \iff \\
 & & \quad (\exists_{w_1 \in \omega}.f_2\ w = (w_1, v_2) \wedge f_1\ w_1 = (w_2, v_1))
\end{array}
$$

$$
\begin{array}{rcl}
|||_{\mathfrak{s}}, \mathsf{ally}_{\mathfrak{s}} & : & \alpha \to \alpha \to \mathbb{B} \\
a_l\ |||_{\mathfrak{s}}\ a_r & \triangleq & \forall_{w \in \omega}.\mathtt{wa}\ a_l\ w = \textsc{False} \wedge \mathtt{wa}\ a_r\ w = \textsc{False} \implies \mathsf{cmmt}(\mathtt{af}\ a_l\ , \mathtt{af}\ a_r\ , w) \\
\mathsf{ally}_{\mathfrak{s}}(a_l, a_r) & \triangleq & \forall_{w \in \omega}.\forall_{w_1 \in \omega}.\forall_{v \in \nu}.\mathtt{wa}\ a_l\ w = \textsc{False} \wedge \mathtt{af}\ a_l\ w = (w_1, v) \wedge \\
 & & \quad \neg(\mathtt{wa}\ a_r\ w = \textsc{True}) \implies \mathtt{wa}\ a_r\ w\ =\ \mathtt{wa}\ a_r\ w_1
\end{array}
$$

$$
\begin{array}{rcl}
\sqsubseteq_{\mathfrak{s}}, \diamondsuit_{\mathfrak{s}} & : & \varsigma \to \varsigma \to \mathbb{B} \\
c_1 \sqsubseteq_{\mathfrak{s}} c_2 & \triangleq & \forall_{a \in \alpha}.\mathtt{ap}\ c_1\ a \implies \mathtt{ap}\ c_2\ a \\
c_l \diamondsuit_{\mathfrak{s}} c_r & \triangleq & \forall_{a_l \in \alpha}.\forall_{a_r \in \alpha}.\mathtt{ap}\ c_l\ a_l \wedge \mathtt{ap}\ c_r\ a_r \implies a_l\ |||_{\mathfrak{s}}\ a_r \wedge \mathsf{ally}_{\mathfrak{s}}(a_l, a_r) \wedge \mathsf{ally}_{\mathfrak{s}}(a_r, a_l)
\end{array}
$$

Figure 2.1: Relations on actions and contexts

Figure 2.1 defines two important relations on contexts.

- $c_1 \sqsubseteq_{\mathfrak{s}} c_2$: any action permitted by context c_1 is also permitted by c_2.
- $c_1 \diamondsuit_{\mathfrak{s}} c_2$: for all actions a_1 and a_2 permitted by contexts c_1 and c_2 respectively, it is true that $a_1\ |||_{\mathfrak{s}}\ a_2$, $\mathsf{ally}_{\mathfrak{s}}(a_1, a_2)$ and $\mathsf{ally}_{\mathfrak{s}}(a_2, a_1)$.

$\diamondsuit_{\mathfrak{s}}$ is symmetric and $\sqsubseteq_{\mathfrak{s}}$ is a pre-order – both by definition.

2.1.3 Enforcing determinism – `pf`

Let us say that a process running in context c forks into two processes running in contexts c_l and c_r. To guarantee deterministic behaviour:

- The order in which actions in the two child sub-programs are performed should be irrelevant. That is to say: if the runtime system can make a choice between doing an action a_l permitted by c_l or an action a_r permitted by c_r, then neither action can impede the other causing it to become stalled, and when both are finally executed their order should not affect the resultant world state or the actions' return values: $c_l \diamondsuit_{\mathfrak{s}} c_r$.
- No child process should be allowed to perform an action forbidden by the parent context: $c_l \sqsubseteq_{\mathfrak{s}} c$ and $c_r \sqsubseteq_{\mathfrak{s}} c$.

These properties are usually undecidable. To solve this problem, and give an actual language implementation, we assume the existence of a function which has been proved to

$$\mathsf{PRE}_{\mathfrak{s}} \triangleq \forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\mathtt{pf}\ p\ c = (c_l,c_r) \Longrightarrow c_l \sqsubseteq_{\mathfrak{s}} c \wedge c_r \sqsubseteq_{\mathfrak{s}} c \wedge c_l \lozenge_{\mathfrak{s}} c_r$$

Figure 2.2: The pre-condition $\mathsf{PRE}_{\mathfrak{s}}$

$$\mathfrak{bfft} :: \mathsf{IOModel}\ \nu_{\mathtt{Bffr}}\ \alpha_{\mathtt{Bffr}}\ \rho_{\mathtt{Bffr}}\ \omega_{\mathtt{Bffr}}\ \varsigma_{\mathtt{Bffr}} \triangleq \langle \mathtt{af}_{\mathtt{Bffr}}, \mathtt{wa}_{\mathtt{Bffr}}, \mathtt{ap}_{\mathtt{Bffr}}, \mathtt{pf}_{\mathtt{Bffr}}\rangle$$

$$\nu_{\mathtt{Bffr}} \triangleq \mathsf{Int} \qquad \alpha_{\mathtt{Bffr}} \triangleq \textsc{Send}\ \mathsf{Int} \mid \textsc{Rcve} \qquad \omega_{\mathtt{Bffr}} \triangleq [\mathsf{Int}]$$
$$\rho_{\mathtt{Bffr}} \triangleq \mathsf{Bool} \qquad \varsigma_{\mathtt{Bffr}} \triangleq \textsc{TopC} \mid \textsc{SubC}\ \mathsf{Bool}$$

$$\begin{array}{llll}
\mathtt{af}_{\mathtt{Bffr}} & (\textsc{Send}\ i) & is & \triangleq\ (is +\!\!+ [i], 0) \\
\mathtt{af}_{\mathtt{Bffr}} & \textsc{Rcve} & (i : is) & \triangleq\ (is, i) \\
\mathtt{wa}_{\mathtt{Bffr}} & (\textsc{Send}\ i) & is & \triangleq\ \textsc{False} \\
\mathtt{wa}_{\mathtt{Bffr}} & \textsc{Rcve} & is & \triangleq\ \mathtt{null}\ is
\end{array}$$

$$\begin{array}{llll}
\mathtt{ap}_{\mathtt{Bffr}} & \textsc{TopC} & a & \triangleq\ \textsc{True} \\
\mathtt{ap}_{\mathtt{Bffr}} & (\textsc{SubC}\ b) & (\textsc{Send}\ i) & \triangleq\ b \\
\mathtt{ap}_{\mathtt{Bffr}} & (\textsc{SubC}\ b) & \textsc{Rcve} & \triangleq\ \mathtt{not}\ b
\end{array}$$

$$\mathtt{pf}_{\mathtt{Bffr}}\ b\ \textsc{TopC} \triangleq (\textsc{SubC}\ b, \textsc{SubC}\ (\mathtt{not}\ b))$$

Figure 2.3: $\mathfrak{bfft}$ – a 1-to-1 communication buffer

obey these exact properties. The function $\mathtt{pf} :: \rho \to \varsigma \to (\varsigma,\varsigma)$ splits a context returning two new ones for the two concurrent left and right sub-programs. Elements of the type ρ give the programmer some flexibility with regard to how he or she wishes the current context to be split. $\mathsf{PRE}_{\mathfrak{s}}$, as defined formally in Figure 2.2, is the pre-condition that $\mathtt{pf}$ must obey. We show in Chapter 4 that if $\mathsf{PRE}_{\mathfrak{s}}$ holds then any program whose I/O API is defined by model $\mathfrak{s}$ will be deterministic.

2.2 Five examples

To give an idea of the flexibility of this approach, this section contains a few examples of complete I/O models, all of which obey $\mathsf{PRE}_{\mathfrak{s}}$. All lemmas have been machine-verified, and actual example programs for these models will be given in Chapter 3.

2.2.1 Communication buffers

The I/O model $\mathfrak{bfft}$, defined in Figure 2.3 is a simple 1-1 communication buffer. World state is of type $[\mathsf{Int}]$, and there are two actions: Send i places i at the end of the list (returning 0, a token value); Rcve removes the first element from the list and returns it. If the current context is SubC True one can only send, if it is SubC False one can only receive, and if it is TopC both are permitted. One can only split the context if it is TopC. This will give one sub-program the right to send, and the other the right to receive.

Lemma 2.2.1. $\mathsf{PRE}_{\mathfrak{bfft}}$.

Proof. If $\mathtt{pf}_{\mathtt{Bffr}}$ splits TopC into SubC False and SubC True:

$$\mathfrak{lock} :: \mathsf{IOModel}\ \nu_{\mathtt{Lock}}\ \alpha_{\mathtt{Lock}}\ \rho_{\mathtt{Lock}}\ \omega_{\mathtt{Lock}}\ \varsigma_{\mathtt{Lock}} \triangleq \langle \mathtt{af}_{\mathtt{Lock}}, \mathtt{wa}_{\mathtt{Lock}}, \mathtt{ap}_{\mathtt{Lock}}, \mathtt{pf}\, s_{\mathtt{Lock}} \rangle$$

$$\nu_{\mathtt{Lock}} \triangleq () \qquad \omega_{\mathtt{Lock}} \triangleq \mathsf{Bool}$$
$$\rho_{\mathtt{Lock}} \triangleq () \qquad \varsigma_{\mathtt{Lock}} \triangleq \mathsf{Bool}$$
$$\alpha_{\mathtt{Lock}} \triangleq \textsc{Lock} \mid \textsc{Unlock} \mid \textsc{Wait}$$

$$\begin{array}{llrcl}
\mathtt{af}_{\mathtt{Lock}} & \textsc{Lock} & b & \triangleq & (\textsc{True}, ()) \\
\mathtt{af}_{\mathtt{Lock}} & \textsc{Unlock} & b & \triangleq & (\textsc{False}, ()) \\
\mathtt{af}_{\mathtt{Lock}} & \textsc{Wait} & b & \triangleq & (b, ()) \\
\mathtt{pf}_{\mathtt{Lock}} & () & c & \triangleq & (\textsc{False}, \textsc{False})
\end{array}$$

$$\mathtt{wa}_{\mathtt{Lock}}\ a\ b \triangleq \begin{cases} b, & a = \textsc{Wait} \\ \textsc{False}, & \text{otherwise} \end{cases}$$

$$\mathtt{ap}_{\mathtt{Lock}}\ b\ a \triangleq \begin{cases} b, & a = \textsc{Lock} \\ \textsc{True}, & \text{otherwise} \end{cases}$$

Figure 2.4: $\mathfrak{lock}$ – a mutex

- ($\textsc{SubC False}$) $\Diamond_{\mathfrak{bfft}}$ ($\textsc{SubC True}$): Only a send and a receive can be performed. $\textsc{Rcve} \mid\mid\mid_{\mathfrak{bfft}} (\textsc{Send}\ i)$ holds, because if the receive isn't stalled, the buffer must be non-empty, and therefore the actions must affect different parts of the buffer and be order independent. $\mathsf{ally}_{\mathfrak{bfft}}(\textsc{Rcve}, \textsc{Send}\ i)$ is trivial because a send is always non-stalled. $\mathsf{ally}_{\mathfrak{bfft}}(\textsc{Send}\ i, \textsc{Rcve})$ is true because adding an element to a buffer cannot cause a receive to become stalled.

- ($\textsc{SubC False}$) $\sqsubseteq_{\mathfrak{bfft}}$ $\textsc{TopC}$ and ($\textsc{SubC True}$) $\sqsubseteq_{\mathfrak{bfft}}$ $\textsc{TopC}$: trivial ($\textsc{TopC}$ does not forbid any actions).

□

2.2.2 Many-to-many mutexes

I/O model $\mathfrak{lock}$, defined in Figure 2.4, allows many processes to be synchronised. World state is of type Bool indicating whether the lock is set. There are three actions: $\textsc{Lock}$ sets world state to $\textsc{True}$; $\textsc{Unlock}$ sets world state to $\textsc{False}$; $\textsc{Wait}$ will stall until world state is $\textsc{False}$, then proceed leaving it unchanged. Contexts are either "$\textsc{True}$", meaning all actions are permitted, or "$\textsc{False}$", meaning just "$\textsc{Unlock}$" and "$\textsc{Wait}$" are allowed.

Contexts can be split as many times as one likes, but the child context will always be "$\textsc{False}$". This means that although a program's context *can* be $\textsc{True}$ at the top-level, allowing "$\textsc{Lock}$" to be performed, "$\textsc{Lock}$" can never be performed concurrently with any other action. The world state doesn't keep track of how many processes are waiting for the mutex to be released, so if it was released then locked again, a waiting process might miss this event.

There are no (useful) return values in this I/O model.

Lemma 2.2.2. $\mathsf{PRE}_{\mathfrak{lock}}$.

Proof. The function $\mathtt{pf}_{\mathtt{Lock}}$ splits any context into $\textsc{False}$ and $\textsc{False}$:

$$\mathfrak{ivar} :: \mathsf{IOModel}\ \nu_{\mathtt{IVar}}\ \alpha_{\mathtt{IVar}}\ \rho_{\mathtt{IVar}}\ \omega_{\mathtt{IVar}}\ \varsigma_{\mathtt{IVar}} \triangleq \langle \mathtt{af}_{\mathtt{IVar}}, \mathtt{wa}_{\mathtt{IVar}}, \mathtt{ap}_{\mathtt{IVar}}, \mathtt{pf}_{\mathtt{IVar}} \rangle$$

$$\begin{array}{llll} \nu_{\mathtt{IVar}} \triangleq \mathsf{Int} & \rho_{\mathtt{IVar}} \triangleq \textsc{LeftWr} \mid \textsc{RightWr} \mid \textsc{BothRd} & \alpha_{\mathtt{IVar}} \triangleq \textsc{ReadI} \\ \omega_{\mathtt{IVar}} \triangleq \mathsf{Int} & \varsigma_{\mathtt{IVar}} \triangleq \textsc{NoneC} \mid \textsc{ReadC} \mid \textsc{WriteC} & \quad\quad \mid \textsc{WriteI}\ \mathsf{Int} \end{array}$$

$$\begin{array}{llllll} \mathtt{af}_{\mathtt{IVar}} & \textsc{ReadI}\ i & \triangleq & (i,i) & \mathtt{ap}_{\mathtt{IVar}} \quad \textsc{NoneC} & a \triangleq \textsc{False} \\ \mathtt{af}_{\mathtt{IVar}} & (\textsc{WriteI}\ i_1) & \triangleq & (i_1,0) & \mathtt{ap}_{\mathtt{IVar}} \quad \textsc{ReadC} & (\textsc{WriteI}\ i_1) \triangleq \textsc{False} \\ & & & & \mathtt{ap}_{\mathtt{IVar}} \quad \textsc{ReadC} & \textsc{ReadI} \triangleq \textsc{True} \\ \mathtt{wa}_{\mathtt{IVar}} & a\ i & \triangleq & \textsc{False} & \mathtt{ap}_{\mathtt{IVar}} \quad \textsc{WriteC} & a \triangleq \textsc{True} \end{array}$$

$$\begin{array}{llll} \mathtt{pf}_{\mathtt{IVar}} & p & \textsc{NoneC} & \triangleq (\textsc{NoneC},\textsc{NoneC}) \\ \mathtt{pf}_{\mathtt{IVar}} & p & \textsc{ReadC} & \triangleq (\textsc{ReadC},\textsc{ReadC}) \\ \mathtt{pf}_{\mathtt{IVar}} & \textsc{LeftWr} & \textsc{WriteC} & \triangleq (\textsc{WriteC},\textsc{NoneC}) \\ \mathtt{pf}_{\mathtt{IVar}} & \textsc{RightWr} & \textsc{WriteC} & \triangleq (\textsc{NoneC},\textsc{WriteC}) \\ \mathtt{pf}_{\mathtt{IVar}} & \textsc{BothRd} & \textsc{WriteC} & \triangleq (\textsc{ReadC},\textsc{ReadC}) \end{array}$$

Figure 2.5: $\mathfrak{ivar}$ – a shared integer variable

- Proving $\textsc{False}\ \Diamond_{\mathfrak{lock}}\ \textsc{False}$: The combinations of actions are (1) two unlocks, (2) two waits or (3) an unlock and a wait. $\textsc{Unlock} \mid\mid\mid_{\mathfrak{lock}} \textsc{Unlock}$ holds and so does $\mathsf{ally}_{\mathfrak{lock}}(\textsc{Unlock}, \textsc{Unlock})$ because Unlock changes state in the same way, and is never stalled. $\textsc{Wait} \mid\mid\mid_{\mathfrak{lock}} \textsc{Wait}$ and $\mathsf{ally}_{\mathfrak{lock}}(\textsc{Wait}, \textsc{Wait})$ is true since Wait doesn't affect world state, is order independent and also cannot cause any action to become stalled. $\textsc{Unlock} \mid\mid\mid_{\mathfrak{lock}} \textsc{Wait}$, $\mathsf{ally}_{\mathfrak{lock}}(\textsc{Unlock}, \textsc{Wait})$ and $\mathsf{ally}_{\mathfrak{lock}}(\textsc{Wait}, \textsc{Unlock})$: If both unlocking and waiting are non-stalled, then their order is irrelevant since wait doesn't change the world state – nothing can cause unlock to become stalled, and no amount of unlocking can cause a wait to become stalled.

- $\textsc{False} \sqsubseteq_{\mathfrak{lock}} b$ for all b: True by reflexivity if $b = \textsc{False}$, and if $b = \textsc{True}$ then it is true because context "True" does not forbid any actions.

□

2.2.3 An integer variable

The I/O model $\mathfrak{ivar}$ (Figure 2.5) is an integer variable which can be written to and read using the actions ReadI and WriteI. Neither of these can ever be stalled.

The context is either NoneC (no actions are permitted), ReadC (only reading is allowed) or WriteC (reading and writing are both allowed). When splitting a context, if the context is NoneC or ReadC then the parameter is ignored and the left and right context remain the same as that of the parent. When splitting context "WriteC", however, depending on which of "LeftWr", "RightWr" or "BothRd" is given as parameter, either one side can be allowed to do everything leaving nothing for the other side, or both sides can be allowed to only read the integer.

$$\mathfrak{istr} :: \mathsf{IOModel}\ \nu_{\mathtt{IStr}}\ \alpha_{\mathtt{IStr}}\ \rho_{\mathtt{IStr}}\ \omega_{\mathtt{IStr}}\ \varsigma_{\mathtt{IStr}} \triangleq \langle \mathtt{af}_{\mathtt{IStr}}, \mathtt{wa}_{\mathtt{IStr}}, \mathtt{ap}_{\mathtt{IStr}}, \mathtt{pf}_{\mathtt{IStr}} \rangle$$

$$\alpha_{\mathtt{IStr}} \triangleq \textsc{ReadI} \mid \textsc{WriteI}\ \mathsf{Int} \qquad \rho_{\mathtt{IStr}} \triangleq () \qquad \nu_{\mathtt{IStr}} \triangleq \mathsf{Int}$$
$$\omega_{\mathtt{IStr}} \triangleq \textsc{Empty} \mid \textsc{Full}\ \mathsf{Int} \qquad \varsigma_{\mathtt{IStr}} \triangleq ()$$

$$\mathtt{af}_{\mathtt{IStr}}\ \ \textsc{ReadI}\ \ (\textsc{Full}\ i) \triangleq (\textsc{Full}\ i, i) \qquad \mathtt{wa}_{\mathtt{IStr}}\ \ \textsc{ReadI}\ \ w \triangleq w == \textsc{Empty}$$
$$\mathtt{af}_{\mathtt{IStr}}\ \ (\textsc{WriteI}\ i)\ \ \textsc{Empty} \triangleq (\textsc{Full}\ i, 0) \qquad \mathtt{wa}_{\mathtt{IStr}}\ \ (\textsc{WriteI}\ i)\ \ w \triangleq \textsc{False}$$

$$\mathtt{ap}_{\mathtt{IStr}}\ c\ a \triangleq \textsc{True} \qquad \mathtt{pf}_{\mathtt{IStr}}\ p\ c \triangleq ((), ())$$

Figure 2.6: $\mathfrak{istr}$ – an integer I-structure

Lemma 2.2.3. $\mathsf{PRE}_{\mathfrak{ivar}}$.

Proof. Since no action can be stalled, the $\mathsf{ally}_{\mathfrak{ivar}}$ property holds automatically for any pair of actions.

- Proving $\Diamond_{\mathfrak{ivar}}$: the function $\mathtt{pf}_{\mathtt{IVar}}$ can split contexts in three different ways.

  NoneC $\Diamond_{\mathfrak{ivar}}$ NoneC and WriteC $\Diamond_{\mathfrak{ivar}}$ NoneC are trivially true, since no actions are permitted by NoneC. With ReadC $\Diamond_{\mathfrak{ivar}}$ ReadC, only "ReadI" is allowed, and ReadI $|||_{\mathfrak{ivar}}$ ReadI is true since it doesn't change the world state.

- $c_1 \sqsubseteq_{\mathfrak{ivar}} c_2$ for all c_1, c_2 as split by $\mathtt{pf}_{\mathtt{IVar}}$: It is clear that NoneC $\sqsubseteq_{\mathfrak{ivar}}$ ReadC and ReadC $\sqsubseteq_{\mathfrak{ivar}}$ WriteC, so it can be seen directly from the definition of $\mathtt{pf}_{\mathtt{IVar}}$ that when contexts are split this property is always obeyed.

□

2.2.4 Id's I-structures

The dataflow language Id [81] introduced a simple construct called an I-structure. An I-structure is a shared, concurrent, write-once data-structure which does not introduce non-determinism, and I-structures have been used to develop elegant, concurrent algorithms for computing matrices efficiently [8].

The I/O model $\mathfrak{istr}$ in Figure 2.6 models one single integer I-structure. An $\mathfrak{istr}$ may be read multiple times but only ever written to once, and it is this single-writer policy which guarantees determinism. The world state is either Empty, indicating that the structure has not yet been written to, or Full i indicating that it has been written to and its value is i. The two actions are the same as with $\mathfrak{ivar}$: ReadI and WriteI i. An attempt to read from an empty structure will stall until that structure has been filled. A write, however, may only take place when the structure is empty – writing to an already full I-structure results in failure.

No notion of I/O context is necessary when introducing I-structures, which makes the model very simple – Id did not restrict the permissions given to concurrent processes. There is only one I/O context, (), and it permits all actions.

Lemma 2.2.4. $\mathsf{PRE}_{\mathfrak{istr}}$.

Proof. Each action is always permitted, regardless, so we must prove non-interference properties for all combinations of actions.

- ReadI $|||_{\mathfrak{istr}}$ ReadI: for the actions to be non-stalled the structure must be full. Therefore, regardless, of ordering, both reads will return the same integer leaving world state unchanged.
- ReadI $|||_{\mathfrak{istr}}$ WriteI i_2: if the read is non-stalled the structure must be full, so in both action orderings the write will fail, with the same net result – failure.
- WriteI i_1 $|||_{\mathfrak{istr}}$ WriteI i_2: regardless of the world state one of the actions must fail because it tries to write to a full I-structure.
- $\mathsf{ally}_{\mathfrak{istr}}$(ReadI, ReadI): this is true because ReadI cannot change world state.
- $\mathsf{ally}_{\mathfrak{istr}}$(WriteI i, ReadI): if the read is unstalled then the structure will be full, in which case the write won't succeed at all.
- $\mathsf{ally}_{\mathfrak{istr}}$(_, WriteI i): true because a write action is never stalled.

□

2.2.5 Terminal I/O

As a small real world I/O example, consider the model of terminal I/O given in Figure 2.7.

There are two actions: (PutC c) writes character c, and GetC reads a character. The model is not perfect, since it does not capture any interesting temporal properties of `stdin`/`stdout`. There is no notion of absolute time – each outputted `Char` gives rise to 0 or more inputted characters instantaneously. It is adequate, nonetheless, and obeys the simple property that if characters are available for input then outputting characters cannot change that. The model also lets us exploit the fact that `stdin` and `stdout` are usually two separate handles, and we may want a process to wait for input on one whilst another outputs data on the other (this is probably only useful for filtering programs such as `grep`. Interactive programs would usually require a lock-step synchronisation of input and output).

The proof of $\mathsf{PRE}_{\mathfrak{term}}$ is not difficult. It permits the same degree of concurrency as the $\mathfrak{bfft}$ model. Reading may take place in parallel with writing in this semantics since (1) if a `Char` is ready for input then writing a character will not change that and (2) if a character is not available for input then the read will stall until one is. The function $\mathtt{pf}_{\mathtt{Term}}$ guarantees that if many processes are running concurrently then at most one can call PutC and at most one can perform GetC.

The $\mathfrak{term}$ model may be seen, in one sense, as just a generalisation of $\mathfrak{bfft}$. Consider the type TermIO once again. One possible instance of it is `loopBack`, defined as follows:

$$\mathfrak{term} :: \mathsf{IOModel}\ \nu_{\mathtt{Term}}\ \alpha_{\mathtt{Term}}\ \rho_{\mathtt{Term}}\ \omega_{\mathtt{Term}}\ \varsigma_{\mathtt{Term}} \triangleq \langle \mathtt{af}_{\mathtt{Term}}, \mathtt{wa}_{\mathtt{Term}}, \mathtt{ap}_{\mathtt{Term}}, \mathtt{pf}_{\mathtt{Term}} \rangle$$

$$\nu_{\mathtt{Term}} \triangleq \mathsf{Char} \qquad \alpha_{\mathtt{Term}} \triangleq \textsc{PutC}\ \mathsf{Char} \mid \textsc{GetC}$$
$$\omega_{\mathtt{Term}} \triangleq (\mathsf{[Char]}, \mathsf{TermIO}) \qquad \rho_{\mathtt{Term}}, \varsigma_{\mathtt{Term}} \triangleq \textsc{TCxt}\ \mathsf{Bool}\ \mathsf{Bool}$$

$$\mathsf{TermIO} \triangleq \textsc{TermIO}\ (\mathsf{Char} \to (\mathsf{[Char]}, \mathsf{TermIO}))$$

$$\begin{array}{llll}
\mathtt{wa}_{\mathtt{Term}} & (\textsc{PutC}\ c) & (cs, t) & \triangleq \textsc{False} \\
\mathtt{af}_{\mathtt{Term}} & (\textsc{PutC}\ c) & (cs, \textsc{TermIO}\ f) & \triangleq ((cs +\!\!+ \mathtt{fst}\ (f\ c), \mathtt{snd}\ (f\ c)), \text{' '}) \\
\mathtt{wa}_{\mathtt{Term}} & \textsc{GetC} & (cs, t) & \triangleq \mathtt{null}\ cs \\
\mathtt{af}_{\mathtt{Term}} & \textsc{GetC} & ((c : cs), t) & \triangleq ((cs, t), c) \\
 & & & \\
\mathtt{ap}_{\mathtt{Term}} & (\textsc{TCxt}\ b\ _) & (\textsc{PutC}\ c) & \triangleq b \\
\mathtt{ap}_{\mathtt{Term}} & (\textsc{TCxt}\ _\ b) & \textsc{GetC} & \triangleq b
\end{array}$$

$$\mathtt{pf}_{\mathtt{Term}}\ (\textsc{TCxt}\ bp_p\ bg_p)\ (\textsc{TCxt}\ bp\ bg) \triangleq$$
$$(\textsc{TCxt}\ (bp\ \&\&\ bp_p)\ (bg\ \&\&\ bg_p), \textsc{TCxt}\ (bp\ \&\&\ \mathtt{not}\ bp_p)\ (bg\ \&\&\ \mathtt{not}\ bg_p))$$

Figure 2.7: $\mathfrak{term}$ – a model for terminal I/O

```
loopBack :: TermIO
loopBack = TermIO (\c -> ([c],loopBack))
```

This really just models the "semantics" of a user who re-inputs every character outputted to him/her. Therefore it is no different to a communication buffer between the sending and receiving processes.

2.3 Discussion

These small examples give a flavour of the sort of systems one can model. One of the most surprising and unusual aspects of our I/O models, we believe, is that we don't need to explicitly mention communication channels at all. By permitting actions to be temporarily stalled, one then has enough machinery to synchronise processes and safely transfer information via the world state.

The key to guaranteeing determinism is making sure that when a context c is split into c_l and c_r to allow concurrency, an action in one context cannot influence the behaviour of an action in the other context. This means that an action permitted by c_l cannot block an action permitted by c_r (or vice versa) and the order in which they are executed must be irrelevant. This rules out competition for a limited resource. Some typically non-deterministic constructs are multiple-writer streams, multiple-reader streams and shared, mutable variables.

Actions can become stalled, however, like "Rcve" in the buffer example. Roughly speaking, an action a_1 can only cause a (not necessarily different) action a_2 to become

stalled if they can under no circumstances be executed concurrently. So RCVE can cause a successive call to RCVE to become stalled because two RCVEs cannot, themselves, be run concurrently with one another. Also, if an action a_1 can successfully predict whether another action a_2 is going to stall then a_1 cannot be run concurrently with any action which changes whether a_2 is stalled. For example, in the $\mathfrak{bfft}$ model neither of the sub-contexts, which allow just sending or just receiving, could be modified to permit an action which returns whether the buffer is empty. However, the receiving context could allow one to wait until the buffer is non-empty and the sending context could allow one to wait until the buffer is empty, and neither would admit non-determinism.

There are a few interesting sub-classes of actions.

- Observer actions, which never change world state (like "WAIT" and "READI"). If a_1 is an observer action, then for any a_0, $\mathsf{ally}_{\mathfrak{s}}(a_1, a_0)$, and for any two observer actions a_1 and a_2, $a_1 \mid\mid\mid_{\mathfrak{s}} a_2$ holds.
- Actions which always return the same value (like "LOCK", "WRITEI 7", or an action which, say, increments a counter without indicating it's value). If action a_1 is of this form then $a_1 \mid\mid\mid_{\mathfrak{s}} a_1$.
- Actions which can never be stalled (like "SEND i" and "WRITEI i"). If action a_1 is never stalled then for any action a_0, $\mathsf{ally}_{\mathfrak{s}}(a_0, a_1)$.
- Commutative contexts: if it is true that $c \Diamond_{\mathfrak{s}} c$ for some c, (which will be true if there is some p such that $\mathtt{pf}\ p\ c = (c,c)$, as is the case with READC in the $\mathfrak{ivar}$ example, or with $\mathfrak{istr}$) then the actions permitted by c in a monadic setting form a commutative monad. In other words, there are absolutely no constraints on the ordering of actions.

It should also be noted that many "reasonable" properties of I/O models are not always present. These include: a context which permits all actions; a context which permits no actions; a way of splitting a context such that all permissions are given to one side only; a way splitting contexts in a symmetric way (if a context can be split into (c_l, c_r) can it also be split into (c_r, c_l)?) While elegant, these properties weren't necessary for the confluence proof, so for this reason we did not include them. In Chapter 8 we develop a clearer mathematical model of contexts, but this is mostly future work.

Other notable omissions from I/O models are the notion of an initial world state and an initial, or outermost, I/O context. No initial state is given, and for a good reason – a programmer in general has no control over the initial world state in which a program runs (for example, whether a particular file exists). The idea of an initial, outermost I/O context carries more weight – a programmer surely wants to know what initial permissions his/her program will have when it begins. We do not specify these because in terms of proving general language or program properties this is a special case. In a real implementation, a program would either begin with an "all actions" context, or perhaps some suitable modification to this based on dynamic properties of system state unknown at compile time.

2.4 Chapter summary

In this chapter we described in detail an "I/O model" structure. This mathematically models the API, the effect of each action on global state, the possible permissions sets (I/O contexts) which can be associated with processes, and how these are distributed in the presence of concurrency. We defined a pre-condition $\mathsf{PRE}_{\mathfrak{s}}$ which, if it holds, guarantees that concurrent processes cannot interfere with one another. Five complete example models were then given, and we discussed briefly some ramifications of our choice of structure and pre-condition.

In the following chapter this structure is used to give a full semantics to Curio, a functional language with I/O and concurrency primitives.

Chapter 3

Curio – a language for reasoning about I/O

In this chapter we introduce the Curio language which makes use of the I/O models outlined in Chapter 2. This is a small functional language with concurrency, interprocess communication and monadic constructs which together let the user write expressive programs which perform I/O. This expressivity is due to our ability to allow concurrency yet still enforce determinism.

In reality, Curio is less a full language specification than a rigorous semantics for a collection of powerful I/O primitives which can be grafted onto a pure functional language using solely its denotational semantics. This overall approach is not new. In fact, at the outset it bears many similarities to that taken by the Haskell community in Concurrent Haskell [89], the lazy functional language with probably the most fully-fledged semantics for I/O.

> "Our semantics is stratified in two levels: an inner denotational semantics which describes the behaviour of pure terms, while an outer monadic semantics describes the behaviour of `IO` computations." [88]

What makes Curio different to Concurrent Haskell is the nature of this outer operational semantics. Concurrent Haskell's semantics for I/O is in effect a purely *co-inductive* one. I/O actions are represented by opaque labelled transitions in a CCS-style [74] process calculus. Each action is a distinct observable event, and the meaning of a program is solely determined by the order in which the (possibly infinite number of) actions occur.

Curio's semantics is primarily *inductive* rather than co-inductive*. There exists a "world state" which the program interacts with and modifies by performing primitive actions. Therefore it is only the cumulative effect of a finite number of actions over a program's lifetime that is observable. By loosening this notion of observation and ascribing to each action a precise effect on world state, one then can distinguish actions which do and do

*See [57] for a good tutorial on induction and co-induction.

not interfere with one another and give a useful semantics to deterministic concurrency. (It should be noted that infinite observations can still be handled. Later, in Chapter 7, we develop a co-inductive semantics for Curio which describes a program based on how it responds to arbitrary changes to world state made by other concurrent processes.)

The chapter is structured as follows. Section 3.1 introduces the five new primitives informally, giving examples of their behaviour. Section 3.2 follows this with a formal SOS-style single-step semantics for Curio. Section 3.3 gives the metalanguage encoding, formally describes convergence and divergence, and gives a proof of the recursive enumerability of convergence.

3.1 Language primitives and examples

A Curio program is an element of type $\mathsf{Prog}_{\mathfrak{s}}\ \beta$, where I/O model $\mathfrak{s}$ defines the I/O interface of the program, and β is the type of the program's return value. The five I/O primitives are as follows, where the types ν, α and ρ are bound by I/O model $\mathfrak{s}$ of type $\mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma$:

$$
\begin{aligned}
\texttt{(>>=)} &:: \forall\beta.\ \forall\gamma.\ \mathsf{Prog}_{\mathfrak{s}}\ \beta \to (\beta \to \mathsf{Prog}_{\mathfrak{s}}\ \gamma) \to \mathsf{Prog}_{\mathfrak{s}}\ \gamma \\
\texttt{return} &:: \forall\beta.\ \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \\
\texttt{action} &:: \alpha \to \mathsf{Prog}_{\mathfrak{s}}\ \nu \\
\texttt{test} &:: \forall\beta.\ \alpha \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \\
\texttt{par} &:: \forall\beta.\ \forall\gamma.\ \forall\epsilon.\ \rho \to \mathsf{Prog}_{\mathfrak{s}}\ \beta \to \mathsf{Prog}_{\mathfrak{s}}\ \gamma \to (\beta \to \gamma \to \epsilon) \to \mathsf{Prog}_{\mathfrak{s}}\ \epsilon
\end{aligned}
$$

Central to these language primitives is the notion of an I/O context. Each (sub-)program is executed within a context $c \in \varsigma$, and that context dictates which actions that (sub-) program can perform, thereby affecting the program's outcome.

The full semantics is given further on, but very briefly:

- `>>=` and `return` are the familiar monadic functions from Haskell.
- `action` performs a primitive (atomic) action.
- `test` performs one of two programs depending on whether an action is allowed by the current context.
- `par` runs two programs concurrently.

A program is said to be a **value** (or **evaluated**) if it is of the form `return` v, for some v. A program is said to be an **action** if it is of the form `action` a, for some a. Programs of the form `par` $p\ m_l\ m_r$ $(*)$ are abbreviated using an infix notation as $m_l\ |||^p_*\ m_r$.

Of the original types which an I/O model is parameterised by, α and ρ end up being part of the language's syntax, ς is best understood as an internal runtime data-structure in the language implementation, and ω is the world model (i.e. part of the language's semantics).

Curio is a non-strict language – none of the five primitives evaluate their arguments. As we show later, Prog is just a non-strict higher-order algebraic datatype.

We now give example programs using Haskell's do-notation.

3.1.1 Communication buffer examples

The following program attempts to send a list of integers along a communication buffer. The Boolean return value indicates whether the current context permits the Send action.

```
sendInts :: [Int] → Prog_bfft Bool
sendInts []     = return True
sendInts (i:is) =
  test (Send i) (action (Send i) >>= \_ -> sendInts is) (return False)
```

The program `rcveInts` c attempts to retrieve c integers from the communication buffer. If receiving is not permitted by the program's context then it returns `[]`, the empty list.

```
rcveInts :: Int → Prog_bfft [Int]
rcveInts c = test Rcve (rcveInts' c) (return [])
  where rcveInts' c | c<=0      = return []
                    | otherwise = do
                                  i  <- action Rcve
                                  is <- rcveInts' (c-1)
                                  return (i:is)
```

3.1.2 Mutex examples

The program `lockPar` runs two 𝔩𝔬𝔠𝔨 programs in parallel combining the two resultant return values into a tuple. This is completely general since in the 𝔩𝔬𝔠𝔨 model the ρ type has just one element, `()`.

```
lockPar :: Prog_lock β → Prog_lock γ → Prog_lock (β,γ)
lockPar pl pr = par () pl pr (\vl vr -> (vl,vr))
```

3.1.3 Integer variable examples

The following program applies a function f to the integer variable returning True if the context permitted reading and writing.

```
applyFn :: (Int → Int) → Prog_ivar Bool
applyFn f =  test (WriteI 0)
  (do {i <- action ReadI; action (WriteI (f i)); return True})
  (return False)
```

3.1.4 Terminal I/O examples

The function `putStr` writes a string to `stdout`, or fails if that is not allowed.

```
putStr :: String → Prog_term ()
putStr []     = return ()
putStr (c:cs) = action (PutC c) >>= \_ -> putStr cs
```

`stdPermissions` returns a string which indicates what actions the current context permits.

```
stdPermissions :: Prog_term String
stdPermissions = do
  inIO  <- test GetC (return "stdin") (return "no stdin")
  outIO <- test (PutC 'x') (return "stdout") (return "no stdout")
  return (inIO ++ ", " ++ outIO)
```

The program `getPutC c` outputs character `c` whilst concurrently requesting a character from input. The entire program returns the character that was read.

```
getPutC :: Char → Prog_term Char
getPutC c = par
  (TCxt True False) (action (PutC c)) (action GetC) (\_ c1 -> c1)
```

3.2 Semantics

The non-deterministic single-step semantics for Curio can be found in Figure 3.1. It is expressed in the standard SOS [99] style, which is usually the most "obviously correct" way of describing concurrency.

All the reduction rules describe the behaviour of what we call a world/program pair, written $w \Vdash m$, where w is the world and m is the program[†]. This allows one to describe how a program and world state interact over time. The context $c \in \varsigma$ in which a program is run also affects how the program behaves, so reduction rules are annotated with the current context.

There are three reduction relations, $\longrightarrow^c$, $\uparrow^c$ and $\downarrow^c$.

$$
\begin{aligned}
w \Vdash m \longrightarrow^c w' \Vdash m' &\triangleq \text{“}w \Vdash m \text{ can reduce to } w' \Vdash m' \text{ in context } c\text{”} \\
w \Vdash m \uparrow^c &\triangleq \text{“}w \Vdash m \text{ can fail in context } c\text{”} \\
w \Vdash m \downarrow^c &\triangleq \text{“}w \Vdash m \text{ is in normal form in context } c\text{”}
\end{aligned}
$$

We say a world/program pair *is* in normal form, rather than can be in normal form, because, as Lemma 4.3.2 in Chapter 4 shows, despite non-determinism, if a program has converged to normal form then it cannot either fail or reduce. Similarly, if a world/program

[†] $w \Vdash m$ is really just syntactic sugar for the tuple `(w,m)`.

pair can fail or reduce, then it cannot be in normal form. Failure may seem a slightly unusual thing to include, but, as the metalanguage encoding in the next section will show, it is necessary. A single reduction step in Curio can correspond to many individual steps in the operational semantics of the metalanguage. Equivalently, a failed single-step reduction may denote a never-ending sequence of reduction steps or a runtime error in the metalanguage.

Values (programs of the form `return` v) on their own are in normal form. If a world/program pair is in normal form but not a value, then we say it is **stalled**. If this occurs, the convergence to normal form is caused by one or more stalled actions.

`action` a attempts to perform action a. If it isn't permitted by the context, `action` a always fails. If it is permitted, it may fail, be stalled or reduce to the action's return value all depending on the API. If it reduces, it modifies the world state. `test` $a\ m_t\ m_f$ allows a program to query the context in which it is being run. If action a is permitted in the current context then `test` $a\ m_t\ m_f$ executes m_t, otherwise it executes m_f. Usually m_f is a sort of exception handler, returning a value indicating that certain actions were locked out by the current context. Programs of the form m `>>=` f behave in the normal monadic style: m is reduced repeatedly until it is a value `return` v for some v, then $f\ v$ is reduced. If m at any stage fails or becomes stalled, the same will happen to m `>>=` f. Program $\bot$ always fails.

A **base program** refers to any program of the form `return` v, $\bot$, `return` v `>>=` f, `action` a, `test` $a\ m_t\ m_f$ or `return` v_l $|||^p_*$ `return` v_r. The behaviour of these programs is always entirely deterministic for a given context and world. A program m `>>=` f, where m is not a value, in effect behaves exactly as m does for an individual single step. It is concurrency on its own which introduces non-determinism. $m_l\ |||^p_*\ m_r$, executed in context c, runs m_l and m_r in parallel in contexts c_l and c_r respectively, where `pf` $p\ c = (c_l, c_r)$. If two concurrent sub-programs can either reduce or fail, then either side may be chosen arbitrarily. This continues until

- one side fails, causing both programs in parallel to fail.
- m_l and m_r become values `return` v_l and `return` v_r respectively, in which case the parallel execution terminates, becoming the value `return` $v_l * v_r$.
- one side becomes stalled and the other side converges to normal form (that is: it is either a value or also stalled), causing the concurrent execution of both programs to be stalled.

(The side-condition on the parallel reduction rules that $m_l \neq \bot$ and $m_r \neq \bot$ is undesirable but hard to avoid. If we check at the outset whether both the left and right sub-programs are values it makes the implementation *much* simpler. Unfortunately this means forcing both m_l and m_r to an outermost constructor.)

As a quick example, consider the following reduction for I/O model `ivar`. This also shows the distinction between an actual reduction step in our semantics, $\longrightarrow^c$, and denotational

$$\frac{w \Vdash m \uparrow^c}{w \Vdash m \texttt{ >>= } f \uparrow^c} \qquad \frac{w \Vdash m \downarrow^c}{w \Vdash m \texttt{ >>= } f \downarrow^c} \quad (m \text{ not a value})$$

$$w \Vdash \texttt{return } v \texttt{ >>= } f \longrightarrow^c w \Vdash f\ v \qquad \frac{w \Vdash m \longrightarrow^c w_1 \Vdash m_1}{w \Vdash m \texttt{ >>= } f \longrightarrow^c w_1 \Vdash m_1 \texttt{ >>= } f}$$

`ap` c a	`wa` a w	`af` a w	Behaviour of `action` a
$\bot$			$w \Vdash \texttt{action } a \uparrow^c$
False			$w \Vdash \texttt{action } a \uparrow^c$
True	$\bot$		$w \Vdash \texttt{action } a \uparrow^c$
True	False	$\bot$	$w \Vdash \texttt{action } a \uparrow^c$
True	False	(w_1, v)	$w \Vdash \texttt{action } a \longrightarrow^c w_1 \Vdash \texttt{return } v$
True	True		$w \Vdash \texttt{action } a \downarrow^c$

`ap` c a	Behaviour of `test` $a\ m_1\ m_2$
$\bot$	$w \Vdash \texttt{test } a\ m_1\ m_2 \longrightarrow^c w \Vdash \bot$
False	$w \Vdash \texttt{test } a\ m_1\ m_2 \longrightarrow^c w \Vdash m_2$
True	$w \Vdash \texttt{test } a\ m_1\ m_2 \longrightarrow^c w \Vdash m_1$

$$w \Vdash \texttt{return } v \downarrow^c$$

$$w \Vdash \bot \uparrow^c$$

$$\frac{\texttt{pf } p\ c\ =\ \bot}{w \Vdash m_l \texttt{ ||| }^p_* m_r \uparrow^c}$$

$$\texttt{pf } p\ c = (c_l, c_r) \begin{cases} \dfrac{w \Vdash m_l \uparrow^{c_l}}{w \Vdash m_l \texttt{ ||| }^p_* m_r \uparrow^c} \quad \dfrac{w \Vdash m_r \uparrow^{c_r}}{w \Vdash m_l \texttt{ ||| }^p_* m_r \uparrow^c} \\[2ex] w \Vdash \texttt{return } v_l \texttt{ ||| }^p_* \texttt{return } v_r \longrightarrow^c w \Vdash \texttt{return } v_l * v_r \\[2ex] \dfrac{w \Vdash m_l \longrightarrow^{c_l} w' \Vdash m_l'}{w \Vdash m_l \texttt{ ||| }^p_* m_r \longrightarrow^c w' \Vdash m_l' \texttt{ ||| }^p_* m_r} \quad (m_r \neq \bot) \\[2ex] \dfrac{w \Vdash m_r \longrightarrow^{c_r} w' \Vdash m_r'}{w \Vdash m_l \texttt{ ||| }^p_* m_r \longrightarrow^c w' \Vdash m_l \texttt{ ||| }^p_* m_r'} \quad (m_l \neq \bot) \\[2ex] \dfrac{w \Vdash m_l \downarrow^{c_l} \quad w \Vdash m_r \downarrow^{c_r}}{w \Vdash m_l \texttt{ ||| }^p_* m_r \downarrow^c} \quad (m_l, m_r \text{ not both values}) \end{cases}$$

Figure 3.1: Non-deterministic single-step semantics for Curio

equality in the metalanguage, $=$.

$$9 \Vdash (\texttt{action}\ \textsc{ReadI})\ |||^{\textsc{BothRd}}_{(+)}\ (\texttt{action}\ \textsc{ReadI}\ \texttt{>>=}\ \lambda i.\ \texttt{return}\ (i-1))$$
$$\longrightarrow^{\textsc{ReadC}}$$
$$9 \Vdash (\texttt{action}\ \textsc{ReadI})\ |||^{\textsc{BothRd}}_{(+)}\ (\texttt{return}\ 9\ \texttt{>>=}\ \lambda i.\ \texttt{return}\ (i-1))$$
$$\longrightarrow^{\textsc{ReadC}}$$
$$9 \Vdash (\texttt{action}\ \textsc{ReadI})\ |||^{\textsc{BothRd}}_{(+)}\ ((\lambda i.\ \texttt{return}\ (i-1))\ 9)$$
$$=$$
$$9 \Vdash (\texttt{action}\ \textsc{ReadI})\ |||^{\textsc{BothRd}}_{(+)}\ (\texttt{return}\ 8)$$
$$\longrightarrow^{\textsc{ReadC}}$$
$$9 \Vdash (\texttt{return}\ 9)\ |||^{\textsc{BothRd}}_{(+)}\ (\texttt{return}\ 8)$$
$$\longrightarrow^{\textsc{ReadC}}$$
$$9 \Vdash \texttt{return}\ (9+8)$$
$$=$$
$$9 \Vdash \texttt{return}\ 17$$

Single step reduction is non-deterministic, so there are other possible reduction sequences. The final program above is a value, `return` 17, and therefore in normal form. A program can also be in normal form if every action it is trying to perform is stalled. For example, in model $\mathfrak{bfft}$:

$$[] \Vdash \texttt{action}\ \textsc{Rcve} \downarrow^{\textsc{TopC}}$$
and
$$[] \Vdash \texttt{action}\ \textsc{Rcve}\ \texttt{>>=}\ \lambda c.\ \texttt{action}\ \textsc{Rcve} \downarrow^{\textsc{TopC}}$$

3.3 Convergence/divergence and the implementation

3.3.1 Metalanguage encoding

Figure 3.2 contains information about how Curio is encoded in the metalanguage. The full definitions are in Section A.1.

Programs in Curio are elements of the higher-order, non-strict algebraic type $\mathsf{Prog}\ \nu\ \alpha\ \rho$. The use of an algebraic type is necessary since we need to be precise about how there are exactly five ways of constructing a program. As an example, the `putStr` program given earlier would really be written in Core-Clean as

```
putStr :: String → Prog ν_Term α_Term ρ_Term
putStr []     = Ret 'X'
putStr (c:cs) = Bind (Action (PutC c)) (\_ -> putStr cs)
```

$$
\begin{array}{rcl}
\mathsf{Prog}\ \nu\ \alpha\ \rho & \triangleq & \textsc{Bind}\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\nu \to \mathsf{Prog}\ \nu\ \alpha\ \rho) \\
& \mid & \textsc{Ret}\ \nu \\
& \mid & \textsc{Action}\ \alpha \\
& \mid & \textsc{Test}\ \alpha\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\mathsf{Prog}\ \nu\ \alpha\ \rho) \\
& \mid & \textsc{Par}\ \rho\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\mathsf{Prog}\ \nu\ \alpha\ \rho)\ (\nu \to \nu \to \nu)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Reduction}\ \beta & \triangleq & \textsc{Reduct}\ \beta \mid \textsc{Converged} \\
\mathsf{Dir} & \triangleq & \mathrm{L} \mid \mathrm{R} \\
\mathsf{Guess} & \triangleq & [\mathsf{Dir}]
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{next}_{\mathfrak{s}} & :: & \mathsf{Guess} \to \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \to \mathsf{Reduction}\ (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \\
\mathtt{rdce}_{\mathfrak{s}} & :: & \mathsf{Nat} \to [\mathsf{Guess}] \to \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho)
\end{array}
$$

$$
\mathtt{rdce}_{\mathfrak{s}}\ (i+1)\ (g{:}gs)\ c\ (w, m) = (w'', m'')
$$

$$
\Longleftrightarrow
$$

$$
\left(\exists_{w' \in \omega}. \exists_{m' \in \mathsf{Prog}\ \nu\ \alpha\ \rho}. \begin{array}{c} \mathtt{next}_{\mathfrak{s}}\ g\ c\ (w, m) = \textsc{Reduct}\ (w', m') \wedge \\ \mathtt{rdce}_{\mathfrak{s}}\ i\ gs\ c\ (w', m') = (w'', m'') \end{array} \right)
$$

$$
\mathtt{rdce}_{\mathfrak{s}}\ 0\ gs\ c\ (w, m) = (w, m)
$$

$$
\begin{array}{rcl}
w \Vdash m \longrightarrow^{c} w' \Vdash m' & \triangleq & \exists_{g \in \mathsf{Guess}}. \mathtt{next}_{\mathfrak{s}}\ g\ c\ (w, m) = \textsc{Reduct}\ (w', m') \\
w \Vdash m \downarrow^{c} & \triangleq & \exists_{g \in \mathsf{Guess}}. \mathtt{next}_{\mathfrak{s}}\ g\ c\ (w, m) = \textsc{Converged} \\
w \Vdash m \uparrow^{c} & \triangleq & \exists_{g \in \mathsf{Guess}}. \mathtt{next}_{\mathfrak{s}}\ g\ c\ (w, m) = \bot \\
w \Vdash m\ {}^{i}\!\!\longrightarrow\!\!\!\!\!\rightarrow^{c} w' \Vdash m' & \triangleq & \exists_{gs \in [\mathsf{Guess}]}. \mathtt{rdce}_{\mathfrak{s}}\ i\ gs\ c\ (w, m) = (w', m') \\
w \Vdash m \longrightarrow\!\!\!\!\!\rightarrow^{c} w' \Vdash m' & \triangleq & \exists_{i \in \mathbb{N}}. w \Vdash m\ {}^{i}\!\!\longrightarrow\!\!\!\!\!\rightarrow^{c} w' \Vdash m' \\
w \Vdash m\ {}^{i}\!\downarrow\!\!\!\downarrow^{c} w' \Vdash m' & \triangleq & w \Vdash m\ {}^{i}\!\!\longrightarrow\!\!\!\!\!\rightarrow^{c} w' \Vdash m' \quad \wedge \quad w' \Vdash m' \downarrow^{c} \\
w \Vdash m \downarrow\!\!\!\downarrow^{c} w' \Vdash m' & \triangleq & \exists_{i \in \mathbb{N}}. w \Vdash m\ {}^{i}\!\downarrow\!\!\!\downarrow^{c} w' \Vdash m' \\
w \Vdash m\ {}^{i}\!\Downarrow^{c}\ w' \Vdash m' & \triangleq & \forall_{gs \in [\mathsf{Guess}]}. \mathtt{rdce}_{\mathfrak{s}}\ i\ gs\ c\ (w, m) = (w', m') \ \wedge\ w' \Vdash m' \downarrow^{c} \\
w \Vdash m \Downarrow^{c}\ w' \Vdash m' & \triangleq & \exists_{i \in \mathbb{N}}. w \Vdash m\ {}^{i}\!\Downarrow^{c}\ w' \Vdash m' \\
w \Vdash m \Uparrow^{c} & \triangleq & \neg \exists_{w' \in \omega}. \exists_{m' \in \mathsf{Prog}\ \nu\ \alpha\ \rho}. w \Vdash m \downarrow\!\!\!\downarrow^{c} w' \Vdash m'
\end{array}
$$

Figure 3.2: Encoding details for Curio

One consequence is that for our machine-verified proofs the types of `>>=` and `par` become monomorphic. This isn't a serious problem. The property of having only five constructors can be informally guaranteed in a real language using module interfaces, and it is not necessary for a program to be able to query another program's outermost constructor, which is something that algebraic types specifically allow. Furthermore, recent work on Generalised Algebraic Datatypes [107] may permit these rich types within an actual algebraic type. We therefore retain the more flexible types.

Since the datatype is higher-order, our style of operational semantics is a little unusual. Consider the following example program written using the underlying algebraic datatype. It is to be run in I/O model $\mathfrak{ivar}$: if the current context allows a read then it attempts to create an infinite number of read processes; otherwise -1 is returned.

```
let reads = Par BothRd (Action ReadI) reads f
in  Test ReadI reads (Ret -1)
```

Firstly, it appears that `let/in` constructs would also be required as primitive in Curio because all of the I/O-related computation in the above example is contained in these constructs. This is not true. The program above is denotationally equal to the following element of $\mathsf{Prog}\ \nu_{\mathtt{IVar}}\ \alpha_{\mathtt{IVar}}\ \rho_{\mathtt{IVar}}$, whose outermost constructor is Par:

```
Par BothRd (Action ReadI)
    (Par BothRd (Action ReadI)
         (Par BothRd (Action ReadI)
              (...) f) f) f
```

The second reason why the style of operational semantics is unusual is that the above structure is infinite! Since the constructors do not evaluate their arguments, the "syntactic" structure of the term language can be partial or infinite. This is a bit unusual, especially since the confluence result is proved in Chapter 4 by inducting over this very structure. As a result, admissibility constraints must first be satisfied when performing induction.

The function $\mathtt{next}_{\mathfrak{s}}$ implements single-step reduction, and non-determinism is implemented by supplying it with an additional parameter of type Guess. This is a stream of L/R values and it guides the reduction algorithm's search for a redex in the presence of concurrency. We existentially quantify over its values to show that reducing to a particular reduct is possible. If a world/program pair is in normal form in some context then $\mathtt{next}_{\mathfrak{s}}$ returns Converged. Otherwise it returns Reduct `(`w_1`,`m_1`)`, for some w_1, m_1, or fails ($\bot$). These outcomes define the three single-step reduction relations, $\downarrow^c$, $\longrightarrow^c$ and $\uparrow^c$.

3.3.2 Convergence and divergence

To make the move from program reduction to program evaluation we must investigate the repeated single-step reduction of a world/program pair. The non-deterministic operator

$\longrightarrow\!\!\!\!\!\rightarrow^c$ is defined to be the reflexive, transitive closure of $\longrightarrow^c$. Using this we can define a non-deterministic convergence relation $\downarrow\!\!\!\!\downarrow^c$. $w \Vdash m \downarrow\!\!\!\!\downarrow^c w' \Vdash m'$ means "$w \Vdash m$ can, after zero or more single-step reductions in context c yield $w' \Vdash m'$, which is in normal form."

This is a rather weak property. It would be better if we knew that $w \Vdash m$ would *always* reduce to a normal form $w' \Vdash m'$ in some context c. This is expressed as $w \Vdash m \Downarrow^c w' \Vdash m'$. Divergence in Curio, expressed as $w \Vdash m \Uparrow^c$, means that a program can under no circumstances reduce in context c to a world/program pair in normal form. The relationship between $\uparrow$ and $\Uparrow$ is subtle. The former is failure in the denotational semantics of the meta-language, the latter is failure in the operational semantics of our language. Theorem 4.5.1 in Chapter 4 states that if $\mathsf{PRE}_{\mathtt{s}}$, then $w \Vdash m \uparrow^c$ implies $w \Vdash m \Uparrow^c$.

The relation $\downarrow\!\!\!\!\downarrow$ is defined using the function $\mathtt{rdce}_{\mathtt{s}}$. This single-step reduces a world/program pair a specific number of times. It requires a list, or stream of Guesses, one for each single-step reduction (we don't mention any boundary or definedness conditions, but they do exist). Convergence, $\Downarrow$, differs to $\downarrow\!\!\!\!\downarrow$ in that it universally quantifies over the guesses. It is trivially true that $w \Vdash m \Downarrow^c w' \Vdash m'$ implies $w \Vdash m \downarrow\!\!\!\!\downarrow^c w' \Vdash m'$, and the confluence proof shows that if $\mathsf{PRE}_{\mathtt{s}}$ holds, $w \Vdash m \downarrow\!\!\!\!\downarrow^c w' \Vdash m'$ implies $w \Vdash m \Downarrow^c w' \Vdash m'$. Sometimes $\downarrow\!\!\!\!\downarrow$ and $\Downarrow$ are annotated with a number which denotes how many reduction steps took place.

Proposition 3.3.1. *If $w \Vdash m \downarrow\!\!\!\!\downarrow^c w_1 \Vdash m_1$ then it is not the case that $w \Vdash m \Uparrow^c$.*

Proof. Direct from the definition of $\Uparrow$. □

Proposition 3.3.2. *Each of the following statements imply those below it:*

1. $w \Vdash m \Downarrow^c w_1 \Vdash m_1$
2. $w \Vdash m \downarrow\!\!\!\!\downarrow^c w_1 \Vdash m_1$
3. $w \Vdash m \longrightarrow\!\!\!\!\!\rightarrow^c w_1 \Vdash m_1$

Proof. (1) implies (2) since if a property holds for all Guesses it must hold for some Guess. (2) implies (3) because the only extra condition on the former is that $w_1 \Vdash m_1 \downarrow^c$ is in normal form. □

Our single-step semantics can be proved to hold with respect to the encoding. It should also be noted that although this is an implementation in a real language, we have no interest at the moment in its efficiency.

3.3.3 Convergence is recursively enumerable

It is reassuring to note that although convergence is defined in the form of an existential quantification over reduction steps, this is just for convenience.

We now prove separately that $\downarrow\!\!\!\downarrow$ is recursively enumerable. This is a (machine-verified) proof that there exists a function

$$\texttt{run}_{\mathfrak{s}} :: [\mathsf{Guess}] \to \varsigma \to (\omega\,,\mathsf{Prog}\ \nu\ \alpha\ \rho) \to (\omega\,,\mathsf{Prog}\ \nu\ \alpha\ \rho)$$

which implements "reduction to normal form". In other words,

$$\texttt{run}_{\mathfrak{s}}\ gs\ c\ (w\,,m) = (w_1\,,m_1)$$
$$\iff$$
$$\exists_{i\in\mathbb{N}}.\texttt{rdce}_{\mathfrak{s}}\ i\ gs\ c\ (w\,,m) = (w_1\,,m_1)\ \wedge\ w_1 \Vdash m_1 \downarrow^c$$

Once this is proved, convergence and divergence can be expressed in a more intuitive manner as follows:

$$\begin{aligned} w \Vdash m \downarrow\!\!\!\downarrow^c w_1 \Vdash m_1 &\iff \exists_{gs\in[\mathsf{Guess}]}.\texttt{run}_{\mathfrak{s}}\ gs\ c\ (w\,,m) = (w_1\,,m_1) \\ w \Vdash m \Downarrow^c\ w_1 \Vdash m_1 &\iff \forall_{gs\in[\mathsf{Guess}]}.\texttt{run}_{\mathfrak{s}}\ gs\ c\ (w\,,m) = (w_1\,,m_1) \\ w \Vdash m \Uparrow^c &\iff \forall_{gs\in[\mathsf{Guess}]}.\texttt{run}_{\mathfrak{s}}\ gs\ c\ (w\,,m) = \bot \end{aligned}$$

Showing that this holds is a non-trivial task. If one just repeatedly applies $\texttt{next}_{\mathfrak{s}}$, perhaps an infinite number of times, then there is no structure to induct over. We must build an implementation which internally constructs an intermediate list.

The proof relies on a more general (and quite powerful) lemma. Consider the following two functions:

```
    iterate  ::  (β → β) → β → [β]
iterate f x  ≜  x : iterate f (f x)
    dountil  ::  (β → β) → (β → Bool) → β → [β]
dountil f p x ≜ x : if (p x) then [] else (dountil f p (f x))
```

`iterate` is a standard Haskell library function. The term `iterate` f x is an infinite list in which each successive element is the next iteration of function f to initial value x. `dountil` is somewhat similar except there is an extra computable predicate $p :: \beta \to \mathsf{Bool}$ which indicates whether the iteration of f should stop. Therefore `dountil` may or may not return an infinite list.

The following lemma gives a useful relationship between the two. It shows that given side-conditions relating p and f, applying f to x_0 i times yields an x_1 such that $p\ x_1$ if and only if `last` (`dountil` $f\ p\ x_0$) $= x_1$ and x_1 is defined.

Lemma 3.3.1.

$$\begin{aligned}&\forall_{f\in\beta\to\beta}.\forall_{p\in\beta\to \mathit{Bool}}.p\ \bot = \bot\ \wedge\ f\ \bot = \bot\ \wedge\ (\forall_{x'\in\beta}.p\ x' \neq \textsc{False} \Longrightarrow f\ x' = \bot)\\ &\quad\Longrightarrow \forall_{x_0\in\beta}.\forall_{x_1\in\beta}.\\ &\qquad(\exists_{i\in\mathbb{N}}.x_1 = \mathit{iterate}\ f\ x_0\ !!\ i\ \wedge\ p\ x_1)\\ &\qquad\quad\Longleftrightarrow\\ &\qquad(x_1 = \mathit{last}\ (\mathit{dountil}\ f\ p\ x_0)\ \wedge\ x_1 \neq \bot)\end{aligned}$$

Proof. The proof needs quite a few extra lemmas, and on the whole requires an extremely careful treatment of non-termination. In particular, with the given side-conditions

- If $f\ x = \bot$ then `iterate` $f\ x = [x, \bot, \bot, \bot, \cdots]$.
- If $p\ x = \bot$ then `dountil` $f\ p\ x = x : \bot$ and `iterate` $f\ x = [x, \bot, \bot, \bot, \cdots]$.
- If $p\ x = \textsc{True}$ then `dountil` $f\ p\ x = [x]$ and `iterate` $f\ x = [x, \bot, \bot, \bot, \cdots]$.
- If $p\ x = \textsc{False}$ and $f\ x = \bot$ then `dountil` $f\ p\ x = x : (\bot : \bot)$.

To prove the $\Longrightarrow$ direction we induct over i, the number of iterations required. We must show that if an iteration results in $\bot$ then it cannot revert back to a non-$\bot$ term and also prove that if $p\ x = \textsc{False}$ then `last` (`dountil` $f\ p\ x$) = `last` (`dountil` $f\ p\ (f\ x)$).

To prove the $\Longleftarrow$ direction we must prove first that

- if $l =$ `length` (`dountil` $f\ p\ x$) and $l \neq \bot$ then p (`iterate` $f\ x\ !!\ (l-1)$).
- if for all i, $0 \leq i < k$, p (`iterate` $f\ x\ !!\ i$) = `False` then `dountil` $f\ p\ x\ !!\ k =$ `iterate` $f\ x\ !!\ k$.
- various other results to do with `last`, `iterate` and infinite lists.

We can show that if (`dountil` $f\ p\ x$) is infinite then `last` (`dountil` $f\ p\ x$) will be $\bot$ (thus proving a contradiction) and if (`dountil` $f\ p\ x$) is a specific finite length then i will be that length minus one. □

To prove the final result we construct the function $\texttt{nextWrap}_{\mathfrak{s}}$, defined in Section A.1, which treats the state of the program's evolution as a 4-tuple containing (1) the world state, (2) the program, (3) the current fresh list of Guess and (4) a Boolean value indicating whether the previous iteration resulted in a world/program pair in normal form.

$$\texttt{nextWrap}_{\mathfrak{s}}\ ::\ \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho, [\mathsf{Guess}], \mathsf{Bool}) \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho, [\mathsf{Guess}], \mathsf{Bool})$$

Now we can give the implementation of $\texttt{run}_{\mathfrak{s}}$, where `fth4` returns the fourth element from a 4-tuple, and prove, using Lemma 3.3.1, that it implements $\Downarrow^{c}$.

$\texttt{run}_{\mathfrak{s}} :: [\mathsf{Guess}] \to \varsigma \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho) \to (\omega, \mathsf{Prog}\ \nu\ \alpha\ \rho)$
$\texttt{run}_{\mathfrak{s}}\ gs\ c\ (w, m) = \texttt{case}\ (\texttt{last}\ (\texttt{dountil}\ (\texttt{nextWrap}_{\mathfrak{s}}\ c)\ \texttt{fth4}\ (w, m, gs, \textsc{False})))\ \texttt{of}$
$\quad (w_1, m_1, \_, \_) \quad \to \quad (w_1, m_1)$

Proposition 3.3.3. *If* $w \Vdash m \Downarrow^c w_1 \Vdash m_1$ *and* $w \Vdash m \Downarrow^c w_2 \Vdash m_2$ *then it is true that* $m_1 = m_2$ *and* $w_1 = w_2$

Proof. Obvious, since we have shown that reduction to some normal form defines a function, not just a relation. □

3.4 Chapter summary

Making use of the I/O model structure defined in Chapter 2, this chapter consisted of a complete formal definition of the Curio language. This included the semantics of the five Curio primitives, a description of the non-deterministic encoding in the metalanguage Core-Clean, a definition of convergence and divergence in Curio, and a proof of the recursive enumerability of convergence.

The definition of the Curio language in this chapter completes the core, foundational material of this dissertation. From now on, the document branches into various distinct strands, some more practically oriented than others.

- The following chapter, Chapter 4, gives the detailed and largely self-contained proof that $\mathsf{PRE}_{\mathfrak{s}}$ guarantees confluence in Curio.
- Chapters 5 and 6 focus on the construction of a large, real world I/O model.
- Chapters 7 and 8 are concerned with the semantic properties of Curio programs.

Chapter 4

Confluence

4.1 Introduction

A non-deterministic reduction system is said to be confluent if for a given term all possible reduction sequences eventually yield the same normal form, or no normal form at all. In this chapter we prove that reduction in Curio is confluent when $\mathsf{PRE}_{\mathfrak{s}}$ holds – that $\downarrow$ is equivalent to $\Downarrow$. This powerful property means that although our definition of concurrency is a natural, non-deterministic one involving arbitrary choices, this arbitrariness is contained and has no effect on the overall outcome. Therefore, in Curio, for all reduction orders, a program will either always terminate with the same resultant world/program pair or always diverge.

That the confluence proof has been machine-verified is also, on its own, a relatively notable result. Confluence proofs for the λ-calculus have been machine-verified before (in Coq [53], and Isabelle/HOL [82]) but we have yet to see one in an LCF style. Perhaps there is a good reason for this – confluence proofs usually wouldn't require one to prove properties about a *program*. This, however, is the approach we took. We prove that our simple implementation of Curio is confluent.

4.1.1 Terminology

Confluence is also known as the "Church-Rosser" property after the authors of the original proof in 1935 for the λ-calculus [18].

Formal definitions of confluence tend to differ slightly depending on which texts are read. Barendregt, in the standard reference text on the λ-calculus [9], defines a reduction system to be confluent if it obeys the "diamond property". This means that if $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$ then there is some D such that $B \twoheadrightarrow D$ and $C \twoheadrightarrow D$, where $\twoheadrightarrow$ is the reflexive, transitive closure of the reduction relation '$\longrightarrow$'. In Term Rewriting Systems [113], however, the diamond property (and confluence) is defined to be the property "$A \longrightarrow B$ and $A \longrightarrow C$ implies there is some D such that $B \longrightarrow D$ and $C \longrightarrow D$". It is the latter definition of

confluence and the diamond property which is closest to what we use, but because our operational semantics also has a notion of failure this muddies the water somewhat.

There is also a certain amount of confusion concerning the differences between (finite) reduction *sequences* and reduction *strategies*. The pure λ-calculus is confluent, yet certain reduction strategies may, for a given term, not result in a normal form when others do. In particular, given the term $(\lambda x.\lambda y.y)\Omega$, call-by-name reduction will find the normal form $(\lambda y.y)$ but call-by-value reduction will not terminate.

The confluence proof in this chapter is stronger. All reduction strategies will have the same effect. The intuitive reason why this is true is that unlike function application in the λ-calculus, our `par` construct is highly symmetric.

4.1.2 Overview

The full machine-verified proof of confluence is long and full of complicated details. Many of these relate to the propagation of $\bot$ and the fact that induction over lazy structures must be admissible (see Section B.2). We adopt a hybrid approach to describing the proof. We try to explain all the technical problems encountered, while still never losing sight of the overall picture.

To give some structure to the proof, three important sections of the proof each culminate with the proof of a key theorem:

- Theorem 4.3.1: If $c_l \Diamond_{\mathfrak{s}} c_r$ and $\mathsf{pre}_{\mathfrak{s}}$ then reduction of programs in context c_l and c_r, if possible in both contexts, is order independent.
- Theorem 4.5.1: If $\mathsf{PRE}_{\mathfrak{s}}$, then $w \Vdash m \uparrow^c$ implies $w \Vdash m \Uparrow^c$.
- Theorem 4.6.1 (Confluence): If $\mathsf{PRE}_{\mathfrak{s}}$, then $w \Vdash m \downarrow^c w' \Vdash m'$ implies $w \Vdash m \Downarrow^c w' \Vdash m'$.

The single goal of this chapter is to prove Theorem 4.6.1. It should be noted that the vast majority of the technicalities encountered in doing so are of no relevance at all to the rest of this dissertation. If the reader is uninterested in the details then he/she should skip this chapter entirely.

The pre-condition $\mathsf{PRE}_{\mathfrak{s}}$ was defined in Chapter 2 as

$$\mathsf{PRE}_{\mathfrak{s}} \triangleq \forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\mathtt{pf}\ p\ c = (c_l, c_r) \Longrightarrow c_l \sqsubseteq_{\mathfrak{s}} c \wedge c_r \sqsubseteq_{\mathfrak{s}} c \wedge c_l \Diamond_{\mathfrak{s}} c_r$$

Occasionally during the confluence proof (for example, in Theorem 4.3.1) we do not need the full power of $\mathsf{PRE}_{\mathfrak{s}}$, just something a lot weaker. $\mathsf{pre}_{\mathfrak{s}}$, defined as follows, only guarantees one of the two properties.

$$\mathsf{pre}_{\mathfrak{s}} \triangleq \forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\mathtt{pf}\ p\ c = (c_l, c_r) \Longrightarrow c_l \sqsubseteq_{\mathfrak{s}} c \wedge c_r \sqsubseteq_{\mathfrak{s}} c$$

4.2 Preliminaries

Non-deterministic single-step reduction "$\longrightarrow$" is a fine high-level notation for expressing how a world/program pair can change within a given context. Its downside, however, is that it hides some internal details of the reduction and its implementation, and these details are central to the machine-verified proof. For this chapter and only this chapter they must be exposed.

They include:

- The initial Guess which guides the search for a redex.
- The actions that were performed, if any.
- The whereabouts of the redex if one is eventually found.

The initial Guess is usually "hidden" by an existential quantification, but it is still explicit in the implementation. The other two are truly internal, however, and for this reason the implementation $\texttt{next}_\mathfrak{s}$ had to be rewritten as the interaction of three different functions. (Admissibility, as discussed in Section B.2, states that to obtain information about a lazy structure one must do so constructively and write a function which computes it. One is not able to just prove that it exists).

4.2.1 Deconstructing $\texttt{next}_\mathfrak{s}$

The new definition of $\texttt{next}_\mathfrak{s}$ and the types of the three new functions are in Figure 4.1, and their full definitions may be found in Section A.1.

The function $\texttt{nextR}_\mathfrak{s}$ searches for a redex. If it finds one it indicates the Route to that redex and the action it will perform, if any. A Route is just a finite list of L/R values. When searching for a redex, each time a `par` is encountered the next element of the Route indicates whether to look to the left- or right-hand side.

If a redex doesn't perform an action we call it a **silent** redex. The function `advA` $v\ r\ m$ modifies the action at route r in program m by replacing it with the program `return` v. The program $\texttt{advS}_\mathfrak{s}\ c\ r\ m$ modifies the silent redex at route r in program m. The context information c is required so that a program of the form `test` $a\ m_1\ m_2$ can determine which of the two programs must be executed.

We now prove that the two definitions of $\texttt{next}_\mathfrak{s}$ are equivalent, thus allowing us to pick whichever is the more appropriate when proving a lemma.

Lemma 4.2.1. $next_\mathfrak{s} = next'_\mathfrak{s}$

Proof. A long but straightforward induction on program structure. We omit the details, but it is a proof that the three individual functions add up to the single original one. $\texttt{nextR}_\mathfrak{s}$ does all the searching for a redex but never modifies either the world state or the program. The only reason it needs world state w at all is to check if an action is stalled. The four

$$
\begin{array}{rcl}
\mathsf{RxType}\ \alpha & \triangleq & \textsc{Action}\ a \mid \textsc{Silent} \\
\mathtt{nextR_s} & :: & \mathsf{Guess} \to \varsigma \to \omega \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Redex}\ (\mathsf{Route}, \mathsf{RxType}\ \alpha) \\
\mathtt{advS_s} & :: & \varsigma \to \mathsf{Route} \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho \\
\mathtt{advA} & :: & \nu \to \mathsf{Route} \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho
\end{array}
$$

$$
\begin{array}{l}
\mathtt{next'_s}\ g\ c\ (w, m) \triangleq \mathtt{case}\ (\mathtt{nextR_s}\ g\ c\ w\ m)\ \mathtt{of} \\
\quad \textsc{NoRedex} \to \textsc{Converged} \\
\quad \textsc{Redex}\ (r, \textsc{Silent}) \to \textsc{Reduct}\ (w, \mathtt{advS_s}\ c\ r\ m) \\
\quad \textsc{Redex}\ (r, \textsc{Action}\ a) \to \mathtt{case}\ (\mathtt{af}\ a\ w)\ \mathtt{of} \\
\qquad\qquad (w_1, v) \to \textsc{Reduct}\ (w_1, \mathtt{advA}\ v\ r\ m)
\end{array}
$$

Figure 4.1: Definition of $\mathtt{next'_s}$

stages of the reduction of an action redex `action` a are (1) checking if the action is permitted ($\mathtt{ap}\ c\ a = \textsc{True}$), (2) checking if the action is stalled, ($\mathtt{wa}\ a\ w = \textsc{False}$) (3) performing the action (evaluating $\mathtt{af}\ a\ w$ to some (w_1, v)) and (4) updating the program with the value v. Of these, the first two are performed by $\mathtt{nextR_s}$, the third takes place in the "wiring" and the fourth is performed by `advA`. With any silent redex, $\mathtt{nextR_s}$ just finds the redex returning the route, and $\mathtt{advS_s}$ modifies the program itself. □

4.2.2 Annotating single-step reduction

Redexes are inherently slippery things to reason about since there is no obvious type or set which is isomorphic to them in any useful way. We therefore use Guesses to *quantify* over redexes when we're looking for one and Routes to *identify* a redex when we have found one. The function from Guesses to redexes is onto; the function from redexes to Routes is one-to-one.

Guesses and Routes can be confusingly similar, at times. Although they are both just lists of L/R, they are conceptually somewhat different. A Guess is best understood as always being infinite, and Routes as always being finite. So we can always successfully retrieve the head and tail of a Guess, whereas there is the possibility that a Route is empty. Also, we sometimes implicitly "cast" a Route to a Guess by padding the finite list to make it infinite.

The new annotated single-step reduction can be found in Figure 4.2. Reduction and failure is now annotated with

1. either an action a, indicating that it was an action redex, or a $\bullet$, indicating that it was a silent redex.

2. A mapping $g \mapsto r$, where g is the initial guess, and r is the route of the redex.

$$
\begin{aligned}
w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{a} w_1 \Vdash m_1 &\triangleq \mathtt{nextR_s}\ g\ c\ w\ m = \textsc{Redex}\ (r, \textsc{Action}\ a)\ \wedge \\
&\qquad \mathtt{af}\ a\ w = (w_1, v)\ \wedge\ m_1 = \mathtt{advA}\ v\ r\ m \\
w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{\bullet} w_1 \Vdash m_1 &\triangleq \mathtt{nextR_s}\ g\ c\ w\ m = \textsc{Redex}\ (r, \textsc{Silent})\ \wedge \\
&\qquad m_1 = \mathtt{advS_s}\ c\ r\ m\ \wedge\ w = w_1 \\
w \Vdash m\ {}^{g \mapsto r}\!\uparrow^{c}_{a} &\triangleq \mathtt{nextR_s}\ g\ c\ w\ m = \textsc{Redex}\ (r, \textsc{Action}\ a)\ \wedge\ \mathtt{af}\ a\ w = \bot \\
w \Vdash m\ {}^{g \mapsto \bot}\!\uparrow^{c} &\triangleq \mathtt{nextR_s}\ g\ c\ w\ m = \bot \\
w \Vdash m \downarrow^{c} &\triangleq \mathtt{nextR_s}\ g\ c\ w\ m = \textsc{NoRedex}
\end{aligned}
$$

Figure 4.2: Annotating single-step reduction

In the case of failure, a reduction can fail after finding a redex ($g \mapsto r$) or while searching for a redex ($g \mapsto \bot$).

It should be clear that the new notation covers all possible cases and is a consistent extension to the old notation. If the guess, route or redex type is omitted this means there is an implicit existential quantification. $w \Vdash m \longrightarrow^{c}_{a} w' \Vdash m'$ means there exists some Guess g and a route Route r such that $w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{a} w' \Vdash m'$. If an action a or $\bullet$ is omitted from a reduction then we just don't specify whether it was silent or performed an action.

Corollary 4.2.2. *If* $w \Vdash m \longrightarrow^{c}_{\bullet} w_1 \Vdash m_1$ *then* $w = w_1$.

Proof. Immediate. □

Corollary 4.2.3. *If* $w \Vdash m \longrightarrow^{c}_{a} w_1 \Vdash m_1$ *then* $\mathtt{af}\ a\ w = (w_1, v)$, *for some* v.

Proof. Immediate. □

Corollary 4.2.4. *If* $w \Vdash m\ {}^{g \mapsto r}\!\uparrow^{c}_{a}$ *then* $\mathtt{af}\ a\ w = \bot$.

Proof. Immediate. □

4.2.3 A template for inductive proofs

Over the course of the chapter a great many lemmas are proved by inducting over the recursive structure of Prog $\nu\ \alpha\ \rho$, the higher order algebraic type defined in Figure 3.2. They are all, of course, different, but we can give a general shape to many of the proofs, and this will serve as a basic template.

Base programs (those of the form $\mathtt{return}\ v$, $\bot$, $\mathtt{return}\ v\ \mathtt{>>=}\ f$, $\mathtt{action}\ a$, $\mathtt{test}\ a\ m_t\ m_f$ or $\mathtt{return}\ v_l\ \mathtt{|||}^{p}_{*}\ \mathtt{return}\ v_r$) are always entirely deterministic for a given context and world, and never require any "deeper" knowledge, such as an inductive hypothesis. When inducting over Prog $\nu\ \alpha\ \rho$, it is usually relatively easy to prove properties for these programs.

The behaviour of a program $m\ \mathtt{>>=}\ f$, where m isn't a value $\mathtt{return}\ v$, is solely determined by the behaviour of m. This is important because when inducting over Prog $\nu\ \alpha\ \rho$

no inductive hypothesis can be given for f. If m fails, diverges or is in normal form then the same will be true of m `>>=` f.

Programs of the form m_l `|||`$^p_*$ m_r are usually the most troublesome to prove properties about. If m_l and m_r are both values then it is a base term. It always fails if `pf` p $c = \bot$, $m_l = \bot$ or $m_r = \bot$, so we often just omit these simple cases altogether. If neither of the above are true an inductive hypothesis will be needed for m_l or m_r (and occasionally both) and how they behave in their respective contexts. Proving lemmas inductively for programs like this is made easier if we can perform a simple case analysis on whether the left- or right-hand side was reduced. The following lemmas show how, by examining the resultant Route, we can learn whether the reduced redex was on the left- or right-hand side. This is our standard procedure for determining the side in which a reduction took place – doing case analysis on the Guess would not give this sort of information.

Lemma 4.2.5. *If* $w \Vdash m_l$ `|||`$^p_*$ $m_r \xrightarrow{g \mapsto []}_c w_1 \Vdash m_1$ *then* `pf` p c = `(`c_l`,`c_r`)` *for some* c_l, c_r *and for some* v_l, v_r, m_l = `return` v_l, m_r = `return` v_r, $w_1 = w$ *and* m_1 = `return` $v_l * v_r$.

Proof. Since the route is [] the redex cannot be within either m_l or m_r. Therefore m_l and m_r must be values. □

Lemma 4.2.6. *If* m *is not a value,* `nextR`$_\mathfrak{s}$ g c w m `>>=` f = `nextR`$_\mathfrak{s}$ g c w m.

Proof. Immediate from implementation. □

Lemma 4.2.7. *If* m *is not a value, then* $w \Vdash m$ `>>=` $f \xrightarrow{g \mapsto r}_c w' \Vdash m'$ `>>=` f *if and only if* $w \Vdash m \xrightarrow{g \mapsto r}_c w' \Vdash m'$.

Proof. Immediate from Lemma 4.2.6. □

Lemma 4.2.8. *If* `nextR`$_\mathfrak{s}$ (b:g) c w m_l `|||`$^p_*$ m_r = Redex `((`L:r`),`x`)` *then* `pf` p c = `(`c_l`,`c_r`)` *for some* c_l, c_r *and* `nextR`$_\mathfrak{s}$ g c_l w m_l = Redex `(`r`,`x`)`.

Proof. Regardless of the initial direction b of the Guess, if the resultant Route was L then the left-hand side will have been reduced. □

Lemma 4.2.9. *If* `nextR`$_\mathfrak{s}$ (b:g) c w m_l `|||`$^p_*$ m_r = Redex `((`R:r`),`x`)` *then* `pf` p c = `(`c_l`,`c_r`)` *for some* c_l, c_r *and* `nextR`$_\mathfrak{s}$ g c_r w m_r = Redex `(`r`,`x`)`.

Proof. Symmetric to proof of Lemma 4.2.9. □

Lemma 4.2.10. *If* $w \Vdash m_l$ `|||`$^p_*$ $m_r \xrightarrow[g \mapsto r]{(b:g) \mapsto (L:r)}_c w_1 \Vdash m_1$ *then* `pf` p c = `(`c_l`,`c_r`)` *for some* c_l,c_r *and there is a* m'_l *such that* $w \Vdash m_l \xrightarrow{}_{c_l} w_1 \Vdash m'_l$ *and* $m_1 = m'_l$ `|||`$^p_*$ m_r.

Proof. Immediate from Lemma 4.2.9. □

Lemma 4.2.11. *If* $w \Vdash m_l$ `|||`$^p_*$ $m_r \xrightarrow[g \mapsto r]{(b:g) \mapsto (R:r)}_c w_1 \Vdash m_1$ *then* `pf` p c = `(`c_l`,`c_r`)` *for some* c_l,c_r *and there is a* m'_r *such that* $w \Vdash m_r \xrightarrow{}_{c_r} w_1 \Vdash m'_r$ *and* $m_1 = m_l$ `|||`$^p_*$ m'_r.

Proof. Immediate from Lemma 4.2.9. □

4.3 Initial results

Having established some new notation, in this section we are now ready to begin the confluence proof. We prove some important lemmas which describe both how single-step reduction affects world state and what actions single-step reduction can perform.

Lemma 4.3.1. *If* $\mathit{nextR}_{\mathfrak{s}}\ g\ c\ w\ m = \textsc{NoRedex}$ *then for all* g_1, $\mathit{nextR}_{\mathfrak{s}}\ g_1\ c\ w\ m = \textsc{NoRedex}$.

Proof. Induction on m. It is trivial that `return` v is always in normal form, and if $w \Vdash \texttt{action}\ a$ is in normal form it always will be for world state w. $m_l \mid\mid\mid^p_* m_r$ is only in normal form if m_l and m_r are both in normal form and it's not the case that both are values. g_1 can be either $(L{:}g'_1)$ or $(R{:}g'_1)$, for some g'_1, but in each case m_l and m_r will both be in normal form, regardless of g'_1's value (IH), so the same will be true of $m_l \mid\mid\mid^p_* m_r$. □

Lemma 4.3.2. *If* $w \Vdash m \downarrow^c$ *then it is not the case that* $w \Vdash m \longrightarrow^c w' \Vdash m'$ *for some* w',m' *or that* $w \Vdash m \uparrow^c$. *(This is a proof of confluence for programs which require no reduction steps).*

Proof. Immediate from Lemma 4.3.1. □

Lemma 4.3.3. *If* $\mathit{nextR}_{\mathfrak{s}}\ g\ c\ w\ m = \textsc{Redex}\ (r, \textsc{Action}\ a)$ *then* $\mathtt{wa}\ a\ w = \textsc{False}$.

Proof. Induction on m. The only base redex which isn't silent is `action` a, and if this reduces then `wa` $a\ w = \textsc{False}$. This is true, by induction, for any reduction which performs an action. □

Lemma 4.3.4. *If* $w \Vdash m \longrightarrow^c_a w' \Vdash m'$ *then* $\mathtt{wa}\ a\ w = \textsc{False}$.

Proof. Immediate from Lemma 4.3.3. □

Lemma 4.3.5. *If* $\mathsf{pre}_{\mathfrak{s}}$ *and* $\mathit{nextR}_{\mathfrak{s}}\ g\ c\ w\ m = \textsc{Redex}\ (r, \textsc{Action}\ a)$ *then* $\mathtt{ap}\ c\ a = \textsc{True}$.

Proof. Induction on m. Trivial for `action` a, and true by contradiction for silent base redexes. If $m = m_1$ `>>=` f (m not a value), it is trivial from Lemma 4.2.6. If m is $m_l \mid\mid\mid^p_* m_r$ then do case analysis on the resultant route. If it is true that $\mathtt{nextR}_{\mathfrak{s}}\ (b{:}g)\ c\ w\ m_l \mid\mid\mid^p_* m_r = \textsc{Redex}\ ((L{:}r), x)$, then $\mathtt{pf}\ p\ c = (c_l, c_r)$ for some c_l and c_r, and from Lemma 4.2.8, $\mathtt{nextR}_{\mathfrak{s}}\ g\ c_l\ w\ m_l = \textsc{Redex}\ (r, x)$. From IH, conclude $\mathtt{ap}\ c_l\ a = \textsc{True}$, and by $\mathsf{pre}_{\mathfrak{s}}$, $c_l \sqsubseteq_{\mathfrak{s}} c$, and therefore $\mathtt{ap}\ c\ a = \textsc{True}$. The proof for the right-hand side is symmetric. □

Lemma 4.3.6. *If* $\mathsf{pre}_{\mathfrak{s}}$ *and* $w \Vdash m \longrightarrow^c_a w' \Vdash m'$ *then* $\mathtt{ap}\ c\ a = \textsc{True}$.

Proof. Immediate from Lemma 4.3.5. □

The following lemmas relate successful single-step reduction of m on world w to reduction in a completely unrelated world w'. The key is that after reducing successfully we will have the correct route to that redex. The second time around this route is then used as the new guess to guarantee that the same redex is actually found – some actions in m which were stalled in w (and therefore ignored) may have become unstalled in w', and it necessary to supply the exact route to make sure that we never encounter these unstalled actions, or any other redex.

Lemma 4.3.7. *If* $\mathtt{nextR_s}\ g\ c\ w\ m = \textsc{Redex}\ (r, \textsc{Silent})$ *then for all* w', $\mathtt{nextR_s}\ r\ c\ w'\ m = \textsc{Redex}\ (r, \textsc{Silent})$.

Proof. Induction on m. Trivially true for any silent base redex, since they are deterministic. If $m = m_l\ \mathtt{|||}^p_*\ m_r$ and $g = (b{:}g')$, then do case analysis on the route r. If $r = []$, it is a base redex. If $r = (L{:}r')$ then we can derive $\mathtt{nextR_s}\ g'\ c_l\ w\ m_l = \textsc{Redex}\ (r', \textsc{Silent})$ with Lemma 4.2.8. With IH, prove $\mathtt{nextR_s}\ r'\ c_l\ w'\ m_l = \textsc{Redex}\ (r', \textsc{Silent})$, and therefore $\mathtt{nextR_s}\ r\ c\ w'\ m_l\ \mathtt{|||}^p_*\ m_r = \textsc{Redex}\ (r, \textsc{Silent})$. If the right-hand side is reduced, the proof is similar. Programs of the form $m_1\ \mathtt{>>=}\ f$ are proved easily with induction. □

Lemma 4.3.8. *If* $w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{\bullet}\ w \Vdash m'$ *then for all* w', $w' \Vdash m \xrightarrow{r \mapsto r}{}^{c}_{\bullet}\ w' \Vdash m'$.

Proof. Immediate, from Lemma 4.3.7. □

Lemma 4.3.9. *If* $\mathtt{nextR_s}\ g\ c\ w\ m = \textsc{Redex}\ (r, \textsc{Action}\ a)$ *then for all* w_1, *if* $\mathtt{wa}\ a\ w_1 = \textsc{False}$ *then* $\mathtt{nextR_s}\ r\ c\ w_1\ m = \textsc{Redex}\ (r, \textsc{Action}\ a)$.

Proof. Induction on m. Similar to the proof of Lemma 4.3.7 except the only valid base program is $\mathtt{action}\ a$ – all the others reduce silently. Since the action was already performed successfully in the action's local context c_1, which won't have changed, we know that $\mathtt{ap}\ c_1\ a = \textsc{True}$, and because we know $\mathtt{wa}\ a\ w_1 = \textsc{False}$, that action will definitely be returned as a legitimate, unstalled redex. □

Lemma 4.3.10. *If* $w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{a}\ w' \Vdash m'$, *where* $\mathtt{af}\ a\ w = (w', v)$, *then for all* w_1, *if* $\mathtt{wa}\ a\ w_1 = \textsc{False}$ *and* $\mathtt{af}\ a\ w_1 = (w_1', v)$, *then* $w_1 \Vdash m \xrightarrow{r \mapsto r}{}^{c}_{a}\ w_1' \Vdash m'$.

Proof. Immediate, using Lemma 4.3.9. □

Lemma 4.3.11. *If* $w \Vdash m \xrightarrow{g \mapsto r}{}^{c}_{a}\ w' \Vdash m'$, *then for all* w_1, *if* $\mathtt{wa}\ a\ w_1 = \textsc{False}$ *and* $\mathtt{af}\ a\ w_1 = \bot$, *then* $w_1 \Vdash m\ {}^{r \mapsto r}\!\uparrow^{c}_{a}$.

Proof. Immediate, using Lemma 4.3.9. □

Lemma 4.3.12. *If* $w \Vdash m \xrightarrow{g \mapsto r} c\ w' \Vdash m'$, *then* $w \Vdash m \xrightarrow{r \mapsto r} c\ w' \Vdash m'$.

Proof. If the reduction is silent, apply Lemma 4.3.8. If it performs an action a, apply Lemma 4.3.10 – we know from Lemma 4.3.4 that $\mathtt{wa}\ a\ w = \textsc{False}$. □

Lemma 4.3.13. *If* $w \Vdash m \overset{g \mapsto r}{} \uparrow^c_a$, *then* $w \Vdash m \overset{r \mapsto r}{} \uparrow^c_a$.

Proof. Apply Lemma 4.3.11 – we know from Lemma 4.3.3 that $\texttt{wa}\ a\ w = \textsc{False}$. □

We can now prove that reduction in disjoint contexts is order independent.

Theorem 4.3.1. *Assuming* $\mathsf{pre}_\mathfrak{s}$ *and* $c_l \Diamond_\mathfrak{s} c_r$, *then if* $w \Vdash m_l \xrightarrow{g_l \mapsto r_l} c_l\ w_l \Vdash m'_l$ *holds and* $w \Vdash m_r \xrightarrow{g_r \mapsto r_r} c_r\ w_r \Vdash m'_r$ *holds, then either*

- *there exists a* w_2 *such that both* $w_r \Vdash m_l \xrightarrow{r_l \mapsto r_l} c_l\ w_2 \Vdash m'_l$ *and* $w_l \Vdash m_r \xrightarrow{r_r \mapsto r_r} c_r\ w_2 \Vdash m'_r$
- *or it is the case that both* $w_r \Vdash m_l \overset{r_l \mapsto r_l}{} \uparrow^{c_l}$ *and* $w_l \Vdash m_r \overset{r_r \mapsto r_r}{} \uparrow^{c_r}$.

Proof. Case analysis on whether a reduction is silent.

- One is silent, say that of c_l: $w_l = w$, from Corollary 4.2.2, and both succeed. Let $w_2 = w_r$. Use Lemma 4.3.8 to prove that m_l still reduces to m'_l with world state w_r, and use Lemma 4.3.12 to prove that m_r will behave the same on world w with r_r as its guess instead of g_r. (The proof is symmetric if c_r's redex is silent).
- Both are actions, say a_l and a_r: this means (Corollory 4.2.3) that $\texttt{af}\ a_l\ w = (w_l, v_l)$ and $\texttt{af}\ a_r\ w = (w_r, v_r)$. From Lemma 4.3.6 we know $\texttt{ap}\ c_l\ a_l = \textsc{True}$ and $\texttt{ap}\ c_r\ a_r = \textsc{True}$, and from Lemma 4.3.4, we also know that $\texttt{wa}\ a_l\ w = \textsc{False}$ and $\texttt{wa}\ a_r\ w = \textsc{False}$. Now, since $c_l \Diamond_\mathfrak{s} c_r$ holds this means $a_l \mid\mid\mid_\mathfrak{s} a_r$, $\mathsf{ally}_\mathfrak{s}(a_l, a_r)$ and $\mathsf{ally}_\mathfrak{s}(a_r, a_l)$ are true. Using $\mathsf{ally}_\mathfrak{s}(a_l, a_r)$, prove $\texttt{wa}\ a_r\ w_l = \textsc{False}$, using $\mathsf{ally}_\mathfrak{s}(a_r, a_l)$ prove $\texttt{wa}\ a_l\ w_r = \textsc{False}$, and because $c_l \mid\mid\mid_\mathfrak{s} c_r$, either:
  - There exists a w_2 such that $\texttt{af}\ a_l\ w_r = (w_2, v_l)$ and $\texttt{af}\ a_r\ w_l = (w_2, v_r)$. Apply Lemma 4.3.10 to both sides to prove that they both will succeed.
  - $\texttt{af}\ a_l\ w_r = \bot$ and $\texttt{af}\ a_r\ w_l = \bot$. Apply Lemma 4.3.11 to show that they both fail.

□

4.4 Analysing failure

The next important theorem is that given $\mathsf{PRE}_\mathfrak{s}$, if $w \Vdash m \uparrow^c$ then $w \Vdash m \Uparrow^c$. This is effectively a proof that denotational failure in the metalanguage always causes divergence in our language – if a single-step reduction can possibly fail then, despite non-determinism, it is impossible that the program will ever converge to some normal form.

In previous sections we defined two distinct types of failure:

- $w \Vdash m \overset{g \mapsto r}{} \uparrow^c_a$. A redex is found, $\texttt{action}\ a$, but the action a fails.
- $w \Vdash m \overset{g \mapsto \bot}{} \uparrow^c$. The search for a redex fails outright.

$$
\begin{array}{rcl}
\mathsf{Tree}\ \beta & = & \textsc{Branch}\ (\mathsf{Tree}\ \beta)\ (\mathsf{Tree}\ \beta) \mid \textsc{Leaf}\ \beta \\
\mathtt{redexTree_s} & :: & \varsigma \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Tree}\ (\mathsf{Redex}\ (\mathsf{Route}, (\varsigma, \mathsf{RxType}\ \alpha))) \\
\mathtt{redexList_s} & :: & \mathsf{Guess} \to \varsigma \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to [(\mathsf{Redex}\ (\mathsf{Route}, (\varsigma, \mathsf{RxType}\ \alpha)))] \\
\mathtt{check_s} & :: & \omega \to \mathsf{Redex}\ (\mathsf{Route}, (\varsigma, \mathsf{RxType}\ \alpha)) \to \mathsf{Redex}\ (\mathsf{Route}, \mathsf{RxType}\ \alpha) \\
\mathtt{shuffle} & :: & \mathsf{Guess} \to \mathsf{Tree}\ \beta \to \mathsf{Tree}\ \beta \\
\mathtt{preorder} & :: & \mathsf{Tree}\ \beta \to [\beta] \\
\mathtt{mapTree} & :: & (\beta \to \gamma) \to \mathsf{Tree}\ \beta \to \mathsf{Tree}\ \gamma \\
\mathtt{firstRx} & :: & \mathsf{Redex}\ \beta \to \mathsf{Redex}\ \beta \to \mathsf{Redex}\ \beta \\
\mathtt{addDir} & :: & \mathsf{Dir} \to \mathsf{Redex}\ (\mathsf{Route}, \beta) \to \mathsf{Redex}\ (\mathsf{Route}, \beta)
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{nextR'_s}\ g\ c\ w\ m & \triangleq & \mathtt{firstRxW_s}\ w\ \$\ \mathtt{redexList_s}\ g\ c\ m \\
\mathtt{firstRxW_s}\ w & \triangleq & \mathtt{foldl}\ \mathtt{firstRx}\ \textsc{NoRedex} \circ \mathtt{map}\ (\mathtt{check_s}\ w) \\
\mathtt{redexList_s}\ g\ c\ m & \triangleq & \mathtt{preorder}\ \$\ \mathtt{shuffle}\ g\ \$\ \mathtt{redexTree_s}\ c\ m
\end{array}
$$

Figure 4.3: Definition of $\mathtt{nextR'_s}$

If a program fails in the first way, then proving that it will diverge is quite easy (Lemma 4.5.6). If a program fails in the second way, however, it is much more difficult since we are forced, like with $\mathtt{next_s}$, to rewrite $\mathtt{nextR_s}$, breaking the process of searching for a redex into more manageable pieces. Only by doing this can we determine why searching for a redex failed, yielding $\bot$.

4.4.1 Deconstructing $\mathtt{nextR_s}$

There are obvious tensions at play when trying to re-implement $\mathtt{nextR_s}$. A tree structure perfectly describes the structure of redexes: a leaf node represents a definite redex or lack thereof, and a branch indicates two concurrent programs, each of which may have their own redexes. Furthermore, the Guess used to coordinate the search for a redex is most naturally applied to a tree structure. On the other hand, we must search for a redex sequentially. A list is ideal for this, since each element in a list is indexed uniquely and sequentially by integers in a way that elements of a tree cannot be.

Our solution involved the construction of two temporary structures, a tree and a list. By doing this we can then make use of existing theorems on lists and trees. An outline of the new function $\mathtt{nextR'_s}$ is given in Figure 4.3, and the full details may be found in Section A.1.

The first half is the function $\mathtt{redexList_s}$. This constructs a lazy list of potential redexes and has three parts. $\mathtt{redexTree_s}\ c\ m$ first builds a tree in which each leaf node represents either NoRedex, indicating that no reduction can take place, or $\textsc{Redex}\ (r, (c_1, x))$, indicating that there is a potential redex at $\mathsf{Route}\ r$, where c_1 is the local context for that redex.

x can be either SILENT or ACTION a for some a. There are are only two situations where NOREDEX will occur in the tree: (1) if m is a value `return` v at the top-level, and (2) if a sub-program of m is of the form `return` v_l $|||^p_*$ m_r or m_l $|||^p_*$ `return` v_r and m_r/m_l are not values. The function `shuffle` g then rearranges the tree according to Guess g so that the left-to-right ordering of leaves respects the order in which $\texttt{nextR}_\mathfrak{s}$ originally would have scanned for redexes. Pre-ordering this tree with `preorder` then results in a list of potential redexes in the correct order.

$\texttt{firstRxW}_\mathfrak{s}$ is the second half of the $\texttt{nextR}'_\mathfrak{s}$ algorithm, and finds the first legitimate redex in the list of potential redexes. It has two parts. Mapping $\texttt{check}_\mathfrak{s}$ w across the list throws out any action redexes which (1) aren't permitted by their local context or (2) are permitted but are stalled in the current world w. Actions which aren't allowed result in $\perp$ ($\texttt{nextR}_\mathfrak{s}$ should only return valid actions – Lemma 4.3.5) but permitted actions which are currently stalled become NOREDEX. Having eliminated all spurious redexes from the list, we can simply scan the list looking for the first element which isn't NOREDEX. Folding `firstRx` across the list does exactly that.

The types of a few other internal functions are given in Figure 4.3. The function `mapTree` modifies each element of a tree with a given function and `addDir` prepends either a L or a R to the route contained within a redex.

It is useful to note that this re-implementation manages to separate the four arguments of $\texttt{nextR}'_\mathfrak{s}$ into three different sequential transformations. The context c and program m are supplied to $\texttt{redexTree}_\mathfrak{s}$, the tree is shuffled with the Guess g, and the world value w is only needed by $\texttt{check}_\mathfrak{s}$ to see if actions are stalled.

Unlike $\texttt{nextR}_\mathfrak{s}$, $\texttt{nextR}'_\mathfrak{s}$ is not defined directly in a recursive manner. We therefore need to prove that a direct recursive relationship exists. The following lemmas are all required to show that the behaviour of $\texttt{nextR}'_\mathfrak{s}$ for some program can be understood in terms of its behaviour for the sub-programs of that program.

Lemma 4.4.1.

$$
\begin{array}{lrcl}
(i) & \mathit{length} \circ \mathit{preorder} & = & \mathit{length} \circ \mathit{preorder} \circ \mathit{shuffle}\, g \\
(ii) & \mathit{shuffle}\, g \circ \mathit{shuffle}\, g & = & \mathit{id} \\
(iii) & \mathit{shuffle}\, g_1 \circ \mathit{shuffle}\, g_2 & = & \mathit{shuffle}\, g_2 \circ \mathit{shuffle}\, g_1 \\
(iv) & \mathit{shuffle}\, [L, L, ...] & = & \mathit{id} \\
(v) & \mathit{shuffle}\, g \circ \mathit{mapTree}\, f & = & \mathit{mapTree}\, f \circ \mathit{shuffle}\, g \\
(vi) & \mathit{map}\, f \circ \mathit{preorder} & = & \mathit{preorder} \circ \mathit{mapTree}\, f \\
(vii) & (r1\ `\mathit{firstRx}`\ r2)\ `\mathit{firstRx}`\ r3 & = & r1\ `\mathit{firstRx}`\ (r2\ `\mathit{firstRx}`\ r3) \\
(viii) & \mathit{firstRxW}_\mathfrak{s}\ w\ (r_1 +\!\!+ r_2) & = & \mathit{firstRx}\, (\mathit{firstRxW}_\mathfrak{s}\ w\ r_1)\ (\mathit{firstRxW}_\mathfrak{s}\ w\ r_2) \\
(ix) & r & = & r\ `\mathit{firstRx}`\ \text{NOREDEX} \\
(x) & r & = & \text{NOREDEX}\ `\mathit{firstRx}`\ r \\
(xi) & \mathit{check}_\mathfrak{s}\ \omega \circ \mathit{addDir}\, d & = & \mathit{addDir}\, d \circ \mathit{check}_\mathfrak{s}\ \omega \\
(xii) & \mathit{firstRxW}_\mathfrak{s}\ w \circ \mathit{map}\, (\mathit{addDir}\, d) & = & \mathit{addDir}\, d \circ \mathit{firstRxW}_\mathfrak{s}\ w
\end{array}
$$

Proof. All by straightforward induction or case analysis. □

Result (i) in Lemma 4.4.1 is a proof that shuffling a redex tree does not change the number of elements in that tree.

The results (ii)-(iv) in Lemma 4.4.1 show that `shuffle` g, for all g, forms the elements of an Abelian (or commutative) group under the $\circ$ operator. Function composition is always associative, the identity is `shuffle` $[L, L, ...]$ and each element is its own inverse.

Results (vii), (ix) and (x) show that `firstRx` is a monoidal operator with identity NoRedex. The final two results show that `firstRxW`$_\mathfrak{s}$ and `check` do not query the route at which a redex exists. Instead it is passed through these functions unchanged.

Lemma 4.4.2. *If m is not a value then* `redexList`$_\mathfrak{s}$ g c $(m$ `>>=` $f) =$ `redexList`$_\mathfrak{s}$ g c m.

Proof. Immediate. □

Lemma 4.4.3. *If m_l, m_r are not both values and* `pf` p $c =$ `(`c_l`,`c_r`)` *then*

$$\texttt{redexList}_\mathfrak{s}\ (L{:}g)\ c\ (m_l\ \texttt{|||}^p_*\ m_r) =$$
$$\texttt{map}\ (\texttt{addDir}\ L)\ (\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l) +\!\!+ \texttt{map}\ (\texttt{addDir}\ R)\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)$$

Proof. Equational.

$$\begin{gathered}
\texttt{redexList}_\mathfrak{s}\ (L{:}g)\ c\ m_l\ \texttt{|||}^p_*\ m_r \\
= (\texttt{redexList}_\mathfrak{s}\text{, defn.}) \\
\texttt{preorder}\ \$\ \texttt{shuffle}\ (L{:}g)\ \$\ \texttt{redexTree}_\mathfrak{s}\ c\ (m_l\ \texttt{|||}^p_*\ m_r) \\
= (\texttt{redexTree}_\mathfrak{s}\ \text{defn.}) \\
\texttt{preorder}\ \$\ \texttt{shuffle}\ (L{:}g)\ \$\ \textsc{Branch}\ (\texttt{mapTree}\ (\texttt{addDir}\ L)\ \$\ \texttt{redexTree}_\mathfrak{s}\ c_l\ m_l)\ (\cdots) \\
= (\texttt{shuffle}\ \text{defn.}) \\
\texttt{preorder}\ \$\ \textsc{Branch}\ (\texttt{shuffle}\ g\ \$\ \texttt{mapTree}\ (\texttt{addDir}\ L)\ \$\ \texttt{redexTree}_\mathfrak{s}\ c_l\ m_l)\ (\cdots) \\
= (\texttt{preorder}\ \text{defn.}) \\
(\texttt{preorder}\ \$\ \texttt{shuffle}\ g\ \$\ \texttt{mapTree}\ (\texttt{addDir}\ L)\ \$\ \texttt{redexTree}_\mathfrak{s}\ c_l\ m_l) +\!\!+ (\cdots) \\
= (\text{Lemma 4.4.1, (v)}) \\
(\texttt{preorder}\ \$\ \texttt{mapTree}\ (\texttt{addDir}\ L)\ \$\ \texttt{shuffle}\ g\ \$\ \texttt{redexTree}_\mathfrak{s}\ c_l\ m_l) +\!\!+ (\cdots) \\
= (\text{Lemma 4.4.1, (vi)}) \\
(\texttt{map}\ (\texttt{addDir}\ L)\ \$\ \texttt{preorder}\ \$\ \texttt{shuffle}\ g\ \$\ \texttt{redexTree}_\mathfrak{s}\ c_l\ m_l) +\!\!+ (\cdots) \\
= (\texttt{redexList}_\mathfrak{s}\text{, defn.}) \\
\texttt{map}\ (\texttt{addDir}\ L)\ (\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l) +\!\!+ \texttt{map}\ (\texttt{addDir}\ R)\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)
\end{gathered}$$

□

Lemma 4.4.4. *If m_l, m_r are not both values and* `pf` p $c =$ `(`c_l`,`c_r`)` *then*

$$\texttt{redexList}_\mathfrak{s}\ (R{:}g)\ c\ (m_l\ \texttt{|||}^p_*\ m_r) =$$
$$\texttt{map}\ (\texttt{addDir}\ R)\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r) +\!\!+ \texttt{map}\ (\texttt{addDir}\ L)\ (\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l)$$

Proof. Symmetric to that of Lemma 4.4.3. □

Lemma 4.4.5. *If m is not a value then* $\mathit{nextR}'_{\mathfrak{s}}\ g\ c\ w\ m \mathbin{>\!\!>\!\!=} f = \mathit{nextR}'_{\mathfrak{s}}\ g\ c\ w\ m$

Proof. Immediate from Lemma 4.4.2 □

Lemma 4.4.6. *If* $\mathit{pf}\ p\ c = (c_l, c_r)$ *and m_l,m_r are not both values then*

$$\mathit{nextR}'_{\mathfrak{s}}\ (L{:}g)\ c\ w\ m_l \mid\mid\mid^p_*\ m_r =$$
$$\mathit{firstRx}\ (\mathit{addDir}\ L\ (\mathit{nextR}'_{\mathfrak{s}}\ g\ c_l\ w\ m_l))\ (\mathit{addDir}\ R\ (\mathit{nextR}'_{\mathfrak{s}}\ g\ c_r\ w\ m_r))$$

This states that if the the initial Guess *says to look on the left-hand side first, then first search for a redex in m_l, adding an L to any resultant route. If this fails then search for a redex in m_r, adding an R to any resultant route.*

Proof. Equational.

$$\begin{gathered}
\mathtt{nextR}'_{\mathfrak{s}}\ (L{:}g)\ c\ w\ m_l \mid\mid\mid^p_*\ m_r \\
= (\mathtt{nextR}'_{\mathfrak{s}}\ \text{defn.}) \\
\mathtt{firstRxW}_{\mathfrak{s}}\ w\ (\mathtt{redexList}_{\mathfrak{s}}\ (L{:}g)\ c\ (m_l \mid\mid\mid^p_*\ m_r)) \\
= (\text{Lemma 4.4.3}) \\
\mathtt{firstRxW}_{\mathfrak{s}}\ w\ (\mathtt{map}\ (\mathtt{addDir}\ L)\ (\mathtt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l) +\!\!+ \mathtt{map}\ (\mathtt{addDir}\ R)\ \cdots) \\
= (\text{Lemma 4.4.1, (viii)}) \\
\mathtt{firstRx}\ (\mathtt{firstRxW}_{\mathfrak{s}}\ w\ (\mathtt{map}\ (\mathtt{addDir}\ L)\ (\mathtt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l)))\ (\cdots) \\
= (\text{Lemma 4.4.1, (xii)}) \\
\mathtt{firstRx}\ (\mathtt{addDir}\ L\ (\mathtt{firstRxW}_{\mathfrak{s}}\ w\ (\mathtt{redexList}_{\mathfrak{s}}\ g\ c_l\ m_l)))\ (\mathtt{addDir}\ R\ (\cdots)) \\
= (\mathtt{nextR}'_{\mathfrak{s}}\ \text{defn.}) \\
\mathtt{firstRx}\ (\mathtt{addDir}\ L\ (\mathtt{nextR}'_{\mathfrak{s}}\ g\ c_l\ w\ m_l))\ (\mathtt{addDir}\ R\ (\mathtt{nextR}'_{\mathfrak{s}}\ g\ c_r\ w\ m_r))
\end{gathered}$$

□

Lemma 4.4.7. *If* $\mathit{pf}\ p\ c = (c_l, c_r)$ *and m_l,m_r are not both values then*

$$\mathit{nextR}'_{\mathfrak{s}}(R{:}g)\ c\ w\ m_l \mid\mid\mid^p_*\ m_r =$$
$$\mathit{firstRx}\ (\mathit{addDir}\ R\ (\mathit{nextR}'_{\mathfrak{s}}\ g\ c_r\ w\ m_r))\ (\mathit{addDir}\ L\ (\mathit{nextR}'_{\mathfrak{s}}\ g\ c_l\ w\ m_l))$$

Proof. Similar to proof of Lemma 4.4.6. □

We can now prove the key equivalence result.

Lemma 4.4.8. $\mathit{nextR}_{\mathfrak{s}} = \mathit{nextR}'_{\mathfrak{s}}$.

Proof. Induction on program structure. All base terms are deterministic, so the Guess is irrelevant and it is a simple matter of showing that $\mathtt{check}_{\mathfrak{s}}$ and $\mathtt{redexTree}_{\mathfrak{s}}$ together behave the same as $\mathtt{nextR}_{\mathfrak{s}}$. If the program is of the form $m \mathbin{>\!\!>\!\!=} f$, m not a value, then Lemma 4.2.6 and Lemma 4.4.2 together convert $\mathtt{nextR}_{\mathfrak{s}}$ and $\mathtt{nextR}'_{\mathfrak{s}}$ respectively to the form of the IH. If the program is $m_l \mid\mid\mid^p_* m_r$, m_l and m_r not both values, then depending on whether the

$$\mathsf{wf}_{\mathfrak{s}} : \varsigma \to (\mathsf{Prog}\ \nu\ \alpha\ \rho) \to \mathbb{B}$$

$$\mathsf{wf}_{\mathfrak{s}}(c, m) \triangleq \mathtt{length}\ (\mathtt{preorder}\ (\mathtt{redexTree}_{\mathfrak{s}}\ c\ m)) \neq \bot$$

$$\mathsf{wf}_{\mathfrak{s}}(c, \mathtt{return}\ v) \qquad \mathsf{wf}_{\mathfrak{s}}(c, \mathtt{action}\ a) \qquad \mathsf{wf}_{\mathfrak{s}}(c, \mathtt{test}\ a\ m_t\ m_f) \qquad \neg\mathsf{wf}_{\mathfrak{s}}(c, \bot)$$

$$\mathsf{wf}_{\mathfrak{s}}(c, m \mathtt{>>=} f) \iff \mathsf{wf}_{\mathfrak{s}}(c, m)$$

$$\text{if } \mathtt{pf}\ p\ c = \bot, \text{ then } \neg\mathsf{wf}_{\mathfrak{s}}(c, m_l \mid\mid\mid^{p}_{*} m_r)$$

$$\text{if } \mathtt{pf}\ p\ c = (c_l, c_r), \text{ then } \mathsf{wf}_{\mathfrak{s}}(c, m_l \mid\mid\mid^{p}_{*} m_r) \iff (\mathsf{wf}_{\mathfrak{s}}(c_l, m_l) \wedge \mathsf{wf}_{\mathfrak{s}}(c_r, m_r))$$

Figure 4.4: Well-formedness of programs

Guess is of the form $(L{:}g)$ or $(R{:}g)$ use Lemma 4.4.3 or Lemma 4.4.4. With the IH for m_l and m_r one can then show that $\mathtt{nextR}'_{\mathfrak{s}}$ preserves (1) the order in which the implementation of $\mathtt{nextR}_{\mathfrak{s}}$ searches for redexes and (2) how the resultant route, if there is one, has either L or R prepended to it depending on the location of the redex. □

4.4.2 Well-formedness of programs

For the divergence proof we found it necessary to define the notion of a well-formed program with respect to some context. The purpose of this is to distinguish programs which fail as a result of their structure (programs which are not well-formed) and those that fail because of actions and how they interact with world state (programs that are well-formed). The key to the introduction of well-formedness is the proof that if a program is not well-formed then it will always diverge, *even though it is possible that it may never fail after a finite number of steps.* (Failure, as always, means immediate failure, $\uparrow$).

If a program m is well-formed with respect to context c then this is written as $\mathsf{wf}_{\mathfrak{s}}(c, m)$. The definition and some basic properties of well-formedness is given in Figure 4.4 – a program is well-formed if the list of potential redexes returned by $\mathtt{redexTree}_{\mathfrak{s}}$ is of defined length. If $\neg\mathsf{wf}_{\mathfrak{s}}(c, m)$ then we know m is partial (contains $\bot$s), infinite or the context c was not split successfully in m. The properties in Figure 4.4 are all simple consequences of the implementation, requiring just a few basic results such as

$$\mathtt{length}\ (xs +\!\!+ ys) = \bot \iff (\mathtt{length}\ xs = \bot\ \vee \mathtt{length}\ ys = \bot)$$

To explain why we need this new terminology one needs to understand why an attempt to prove the theorem relating failure to divergence would fail without it. Divergence, $\Uparrow$, expresses the property that for a given world/program pair no reduction sequence of finite length can result in convergence to some normal form. The way that one proves that a world/program pair diverges is to prove, by inducting over i, that if it did converge to normal form after i steps one can show a contradiction.

It was originally thought that the following lemma would suffice: if $\mathsf{PRE}_{\mathfrak{s}}$ and $w \Vdash m \longrightarrow^{c} w' \Vdash m'$, then $w \Vdash m \uparrow^{c}$ implies $w' \Vdash m' \uparrow^{c}$. In other words, if a program can possibly fail, then it cannot escape from that possibility by reducing to a different world/program pair. But this proof strategy doesn't hold for some programs which are badly-formed, and the following counter-example shows why: (to be run in model `lock` from Chapter 2, with any context; world state should be TRUE (the mutex is locked); `f` is irrelevant).

```
let waits = par () (action Wait) waits f
in  par () (action Unlock) waits f
```

The program `waits` attempts to create an infinite number of processes, each of which performs a single WAIT action. The entire program above places an UNLOCK in parallel with `waits`. If the mutex is locked and we first look for a redex on the right-hand side then it will always fail ($\uparrow$) – there are an unlimited number of stalled actions within `waits` and the search for a redex simply will not terminate. After executing (`action` UNLOCK), however, each of the infinite number of WAITs are released, and there will always be a redex, and thus no (immediate) failure can occur*. One of the key intermediate results of the following section is that badly-formed programs always diverge (Lemma 4.5.3). This solves the problem with the `waits` counter-example. Since it is badly-formed it always diverges, so is obviously true that if it can possibly fail it will also always diverge.

Some partial or infinite programs *are* well-formed. Take the example used in Chapter 3:

```
let reads = par BothRd (action ReadI) reads f
in  test ReadI reads (return (-1))
```

`reads` itself isn't well-formed, but the entire program, although "syntactically" infinite, is well-formed. It is also worth noting that well-formedness is a property of programs for a given context, not world/program pairs.

4.5 Failure implies divergence

We are now in a position to begin proving the following key theorem.

Theorem 4.5.1. *If* $\mathsf{PRE}_{\mathfrak{s}}$*, then* $w \Vdash m \uparrow^{c}$ *implies* $w \Vdash m \Uparrow^{c}$*.*

*One may ask, though, that if the single-step reduction function is given a Guess which tells it always to first try the right-hand side of a parallel reduction, it would also fail immediately, never observing an action. That might solve it for this example, but the problem runs much deeper. What is of real interest is that when we said it made the theorem unprovable, *we didn't mean it made it false.* The issue is that our denotational, domain theoretic model of computable programs contains denotations of non-existent programs - which is an example of the full abstraction problem [84]. In the above example, to cause it to fail, the right-hand side must be constantly traversed. But what if the side with an infinite number of actions alternates from left to right? The domain of Prog $\nu\ \alpha\ \rho$ admits any infinite sequence of L/R, but, since an infinite list of Bool can encode any real number, and it's well known that certain real numbers are uncomputable, one cannot always compute a guess which would always pick the correct reduction path to guarantee failure. And the guess *must* be computable, because of the admissibility constraints mentioned.

Proof. If $\neg\mathsf{wf}_{\mathfrak{s}}(c, m)$, use Lemma 4.5.3. If $w \Vdash m \overset{g\mapsto r}{\uparrow}{}^{c}_{a}$, use Lemma 4.5.6. Otherwise it must be true that $\mathsf{wf}_{\mathfrak{s}}(c, m)$ and $w \Vdash m \overset{g\mapsto \bot}{\uparrow}{}^{c}$, so use Lemma 4.5.19. □

Its proof is effectively the merging of three separate lemmas, and these three lemmas are proved in the following three respective sub-sections.

4.5.1 Badly-formed programs

We prove that badly-formed programs diverge by showing that a program cannot reduce out of a badly-formed state, and that no badly-formed program is in normal form.

Lemma 4.5.1. *If* $w \Vdash m \downarrow^{c}$ *then* $\mathsf{wf}_{\mathfrak{s}}(c, m)$.

Proof. Induction on m. A value is well-formed, as is a stalled action (or any action, for that matter). The other base terms are not in normal form, so they don't apply. This is easily proved by induction if $m = m' \texttt{ >>= } f$. When $m = m_l \texttt{ ||| }^{p}_{*} m_r$ is in normal form, m_l and m_r must be as well. Therefore, by IH, $\mathsf{wf}_{\mathfrak{s}}(c_l, m_l)$ and $\mathsf{wf}_{\mathfrak{s}}(c_r, m_r)$ and since the concatenation of two finite lists is a finite list, $\mathsf{wf}_{\mathfrak{s}}(c, m)$.

□

Lemma 4.5.2. *If* $\neg\mathsf{wf}_{\mathfrak{s}}(c, m)$ *and* $w \Vdash m \longrightarrow^{c} w' \Vdash m'$, *then* $\neg\mathsf{wf}_{\mathfrak{s}}(c, m')$.

Proof. Induction on m.

- Base terms. The only base program which isn't well-formed is $\bot$, and it cannot reduce successfully, so doesn't apply.
- $m = m_1 \texttt{ >>= } f$: it must be true that $\neg\mathsf{wf}_{\mathfrak{s}}(c, m_1)$, so m_1 is not a value, and $w \Vdash m_1 \longrightarrow^{c} w'_1 \Vdash w'$ so, by the inductive hypothesis, $\neg\mathsf{wf}_{\mathfrak{s}}(c, m'_1)$. Therefore m'_1 cannot be a value either, and $\neg\mathsf{wf}_{\mathfrak{s}}(c, m'_1 \texttt{ >>= } f)$.
- $m = m_l \texttt{ ||| }^{p}_{*} m_r$: since $\neg\mathsf{wf}_{\mathfrak{s}}(c, m_l \texttt{ ||| }^{p}_{*} m_r)$, that means at least one of the following is true: $\neg\mathsf{wf}_{\mathfrak{s}}(c_l, m_l)$ or $\neg\mathsf{wf}_{\mathfrak{s}}(c_r, m_r)$. Now, say m_l contains the reduced redex. This means $w \Vdash m_l \longrightarrow^{c_l} w'_l \Vdash m'_l$ and $m' = m'_l \texttt{ ||| }^{p}_{*} m_r$, for some m'_l. If it was m_r which was badly-formed then the resultant program will still be badly-formed. If it was m_l that was badly formed then, by IH, $\neg\mathsf{wf}_{\mathfrak{s}}(c_l, m'_l)$ and the resultant program remains badly-formed. (The proof is symmetric if m_r was reduced, not m_l).

□

Lemma 4.5.3. *(Part 1 of Theorem 4.5.1). If* $\neg\mathsf{wf}_{\mathfrak{s}}(c, m)$, *then for all* w, $w \Vdash m \Uparrow^{c}$.

Proof. Induction on the number of reduction steps it might take to reduce to normal form. Base case (0): from Lemma 4.5.1, since m is badly-formed it cannot be in normal form. Inductive case: it can't converge to normal form after $i + 1$ steps because after one step it is still badly-formed (Lemma 4.5.2) and it can't converge after i steps (IH). □

4.5.2 Failure of actions

We prove that the failure of an action implies $w \Vdash m \Uparrow^c$ by showing that even if $w \Vdash m$ can also reduce successfully then it can never "escape" from the action that originally failed. This is the property we originally wanted to prove for all programs, before we showed a counter-example. That it is true for actions which fail is the result of the definition of $|||_{\mathfrak{s}}$. It preserves the fact that $\texttt{af}\ a\ w = \bot$ after the world state is modified by another action.

Lemma 4.5.4. *Given* $\mathsf{pre}_{\mathfrak{s}}$ *and* $c_l \lozenge_{\mathfrak{s}} c_r$*, then if* $w \Vdash m_r \overset{g_r \mapsto r_r}{\uparrow^{c_r}_{a_r}}$ *and* $w \Vdash m_l \longrightarrow^{c_l} w_l \Vdash m'_l$*, then* $w_l \Vdash m_r \overset{r_r \mapsto r_r}{\uparrow^{c_r}_{a_r}}$.

Proof. We know that the action a_r failed: $\texttt{af}\ a_r\ w = \bot$. If m_l reduces silently then $w_l = w$, and Lemma 4.3.13 can be used. If m_l instead performs some action a_l then, from Lemma 4.3.4, $\texttt{wa}\ a_l\ w = \textsc{False}$, from Lemma 4.3.6, $\texttt{ap}\ c_l\ a = \textsc{True}$, and from the definition, $\texttt{af}\ a_l\ w = (w_l, v_l)$, for some v_l. With $c_l \lozenge_{\mathfrak{s}} c_r$, then derive $a_l\ |||_{\mathfrak{s}}\ a_r$ and $\mathsf{ally}_{\mathfrak{s}}(a_l, a_r)$. Using these two facts we can prove $\texttt{af}\ a_r\ w_l = \bot$ and $\texttt{wa}\ a_r\ w_l = \textsc{False}$ respectively. Apply Lemma 4.3.11 to show that a_r will fail when executed in world w_l. □

Lemma 4.5.5. *If* $\mathsf{PRE}_{\mathfrak{s}}$*,* $w \Vdash m \overset{g \mapsto r}{\uparrow^c_a}$ *and* $w \Vdash m \longrightarrow^c w' \Vdash m'$*, then* $w' \Vdash m' \overset{r \mapsto r}{\uparrow^c_a}$.

Proof. Induction on m. True by contradiction for all base programs – they are deterministic, and therefore cannot fail *and* successfully reduce. For $m_1\ \texttt{>>=}\ f$ (where m_1 isn't a value) apply IH. If m is of the form $m_l\ \texttt{|||}^p_*\ m_r$ (and m_l and m_r aren't both values) then do case analysis on whether the failure and the reduction were on the same side or different sides. If both failure and success are on the same side, apply the IH. If failure and success are on opposite sides, apply Lemma 4.5.4 to prove that the side that fails will still fail after a successful reduction on the opposite side. (Since reduction can succeed, $\texttt{pf}\ p\ c = (c_l, c_r)$ for some c_l, c_r, and from $\mathsf{PRE}_{\mathfrak{s}}$ this means $\mathsf{pre}_{\mathfrak{s}}$ and $c_l \lozenge_{\mathfrak{s}} c_r$). □

Lemma 4.5.6. *(Part 2 of Theorem 4.5.1). If* $\mathsf{PRE}_{\mathfrak{s}}$*, then if* $w \Vdash m \overset{g \mapsto r}{\uparrow^c_a}$*, then* $w \Vdash m \Uparrow^c$.

Proof. Induction on the number of reduction steps $w \Vdash m$ might take to reach normal form. Base case (0): since $w \Vdash m \uparrow^c$, by Lemma 4.3.2 it can't be in normal form. Inductive case: it can't reach normal form after $i + 1$ steps because, by Lemma 4.5.5, after one step it can still fail, and by IH it can't converge to normal form after i steps. □

4.5.3 Well-formed failure of $\texttt{nextR}_{\mathfrak{s}}$

Finally, we must prove that, assuming $\mathsf{PRE}_{\mathfrak{s}}$ and $\mathsf{wf}_{\mathfrak{s}}(c, m)$, if $w \Vdash m \overset{g \mapsto \bot}{\uparrow^c}$ then $w \Vdash m \Uparrow^c$. The structure to the proof is rather similar to our proof of Lemma 4.5.6 at a high-level but there are considerably more technical details.

The well-formedness of m has a direct consequence: it means that there are a finite list of potential redexes, so we can induct over the length of the list without admissibility constraints. One can therefore prove existential properties by induction, namely that if

$\texttt{nextR}_\mathfrak{s}$ results in $\bot$ then it failed for some *specific* potential redex identified by an integer. Once we have an identifer for problematic redexes we can then prove admissible properties about a lazy structure such as $\mathsf{Prog}\ \nu\ \alpha\ \rho$.

This section of the proof is not pretty and requires the intricate and cumbersome manipulation of individual lists of redexes. (Our approach was influenced very much by the existence of other pre-proved theorems relating to lists). Also, for the amount of effort required, its relevance is disappointingly small. As we show below, if a program fails in the above manner it is for one of two rather uninteresting reasons. We include the proof for absolute completeness and as a testament to the rigour a proof-assistant imposes on its user.

We begin by proving three lemmas relating lists of potential redexes of programs of the form $m_l \mid\mid\mid^p_* m_r$ to that of m_l and m_r.

Lemma 4.5.7. *If* $\texttt{pf}\ p\ c = (c_l, c_r)$, $\mathsf{wf}_\mathfrak{s}(c, m_l \mid\mid\mid^p_* m_r)$ *and*

$$\texttt{redexList}_\mathfrak{s}\ (b{:}g)\ c\ (m_l \mid\mid\mid^p_* m_r)\ !!\ i = \textsc{Redex}\ ((L{:}r), x)$$

then for some i_l, $\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l\ !!\ i_l = \textsc{Redex}\ (r, x)$.

Proof. First, m_l and m_r cannot both be values since this would mean

$$\texttt{redexList}_\mathfrak{s}\ (b{:}g)\ c\ (m_l \mid\mid\mid^p_* m_r) = [\textsc{Redex}\ ([], (c, \textsc{Silent}))]$$

and $(L{:}r) \neq []$. Perform case analysis on b:

- $b = L$: apply Lemma 4.4.3. Because $\mathsf{wf}_\mathfrak{s}(c, m_l \mid\mid\mid^p_* m_r)$, there are a finite, defined number of potential redexes for both m_l and m_r. If $i < \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l)$ then $\texttt{map}\ (\texttt{addDir}\ L)\ \$\ \texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l\ !!\ i = \textsc{Redex}\ ((L{:}r), x)$, so we know that $\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l\ !!\ i = \textsc{Redex}\ (r, x)$, so let $i_l = i$. If $i \geq \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_l\ m_l)$ then $\texttt{map}\ (\texttt{addDir}\ R)\ \$\ \texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r\ !!\ i = \textsc{Redex}\ ((L{:}r), x)$ and this is a contradiction since $\texttt{addDir}\ R$ cannot modify the resultant route so that it becomes $(L{:}r)$.
- $b = R$: apply Lemma 4.4.4. Similar to the above except m_r is searched first. If $i < \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)$ then we can prove a contradiction. Otherwise $i \geq \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)$, so let $i_l = i - \texttt{length}\ (\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r)$.

□

Lemma 4.5.8. *If* $\texttt{pf}\ p\ c = (c_l, c_r)$, $\mathsf{wf}_\mathfrak{s}(c, m_l \mid\mid\mid^p_* m_r)$ *and*

$$\texttt{redexList}_\mathfrak{s}\ (b{:}g)\ c\ (m_l \mid\mid\mid^p_* m_r)\ !!\ i = \textsc{Redex}\ ((R{:}r), x)$$

then for some i_r $\texttt{redexList}_\mathfrak{s}\ g\ c_r\ m_r\ !!\ i_r = \textsc{Redex}\ (r, x)$.

Proof. Similar to the proof of Lemma 4.5.7. □

Lemma 4.5.9. *If* $\mathtt{pf}\ p\ c = (c_l, c_r)$ *and* $\mathsf{wf}_{\mathfrak{s}}(c, m_l\ /\!/\!/^p_*\ m_r)$ *then*

$$\mathit{redexList}_{\mathfrak{s}}\ g\ c\ (m_l\ /\!/\!/^p_*\ m_r)\ !!\ i = \textsc{Redex}\ ([], x)$$

for some i *if and only if* m_l *and* m_r *are both values.*

Proof. If: by contradiction. If m_l and m_r were not both values then, depending on whether g is $(L{:}g')$ or $(R{:}g')$ for some g', use either Lemma 4.4.3 or Lemma 4.4.4 to show that the resultant route must be either $(L{:}r')$ or $(R{:}r')$ for some r'. A route of value [] is impossible.

Only if: both m_l and m_r are values, so by the implementation there is just one silent reduction at route []. □

The next step is to identify how and why $w \Vdash m\ ^{g\mapsto\bot}\!\uparrow^c$. The following three lemmas together show that if $w \Vdash m\ ^{g\mapsto\bot}\!\uparrow^c$ then for some action a in its local context c_1, either

- $\mathtt{ap}\ c_1\ a \neq \textsc{True}$ (that is: $\mathtt{ap}\ c_1\ a = \textsc{False}$ or $\mathtt{ap}\ c_1\ a = \bot$), or
- $\mathtt{ap}\ c_1\ a = \textsc{True}$ and $\mathtt{wa}\ a\ w = \bot$

Lemma 4.5.10. *If a list* rxs *is finite and* $\mathit{firstRxW}_{\mathfrak{s}}\ w\ rxs = \bot$ *then there exists some* i*,* $i \geq 0$*,* $i < \mathit{length}\ rxs$ *such that* $\mathit{check}_{\mathfrak{s}}\ w\ (rxs\ !!\ i) = \bot$*.*

Proof. Induction on the length of rxs. If $rxs = []$, $\mathtt{firstRxW}_{\mathfrak{s}}\ w\ rxs \neq \bot$. If $rxs = [rx : rxs']$ then either $\mathtt{check}_{\mathfrak{s}}\ w\ rx = \bot$ (let $i = 0$), or $\mathtt{firstRxW}_{\mathfrak{s}}\ w\ rxs' = \bot$, in which case let $i = i' + 1$, where i' is the index from the IH. □

Lemma 4.5.11. *If* $\mathsf{wf}_{\mathfrak{s}}(c, m)$ *and* $w \Vdash m\ ^{g\mapsto\bot}\!\uparrow^c$ *then there exists an integer* i *such that* $(\mathit{redexList}_{\mathfrak{s}}\ g\ c\ m)\ !!\ i = \textsc{Redex}\ (r, (c_1, \textsc{Action}\ a))$ *and that checking this redex results in failure:* $\mathit{check}_{\mathfrak{s}}\ w\ (\textsc{Redex}\ (r, (c_1, \textsc{Action}\ a))) = \bot$*.*

Proof. $w \Vdash m\ ^{g\mapsto\bot}\!\uparrow^c$ is defined as $\mathtt{nextR}_{\mathfrak{s}}\ g\ c\ w\ m = \bot$, which is equivalent to $\mathtt{firstRxW}_{\mathfrak{s}}\ w\ (\mathtt{redexList}_{\mathfrak{s}}\ g\ c\ m) = \bot$. Since $\mathsf{wf}_{\mathfrak{s}}(c, m)$ there are only a finite number of potential redexes, so apply Lemma 4.5.10 to prove that there is some i, such that $\mathtt{check}_{\mathfrak{s}}\ w\ (\mathtt{redexList}_{\mathfrak{s}}\ g\ c\ m\ !!\ i) = \bot$. $\mathtt{redexList}_{\mathfrak{s}}\ g\ c\ m\ !!\ i$ must be of the form $\textsc{Redex}\ (r, (c_1, \textsc{Action}\ a))$ because $\mathtt{check}_{\mathfrak{s}}$ never fails for silent redexes or NoRedex. (There is also a separate and entirely uninteresting proof that $\mathtt{redexList}_{\mathfrak{s}}$ cannot return a list with $\bot$ as an element. We omit this completely). □

Lemma 4.5.12. $\mathit{check}_{\mathfrak{s}}\ w\ (\textsc{Redex}\ (r, (c, \textsc{Action}\ a))) = \bot$ *if and only if either* $\mathit{ap}\ c\ a \neq \textsc{True}$ *or* $\mathit{ap}\ c\ a = \textsc{True} \wedge \mathit{wa}\ a\ w = \bot$*.*

Proof. Immediate. □

We temporarily replace any reference to divergence with properties of $\mathtt{redexList}_{\mathfrak{s}}$ and $\mathtt{check}_{\mathfrak{s}}$ and continue the proof in the style of previous sections. Lemma 4.5.14, Lemma 4.5.15 and Lemma 4.5.16 are really just re-workings of Lemma 4.3.6, Lemma 4.5.4 and Lemma 4.5.5 respectively.

Lemma 4.5.13. *If* $\texttt{check}_\mathfrak{s}\ w\ (\textsc{Redex}\ (r,x)) = \bot$ *then* $\texttt{check}_\mathfrak{s}\ w\ (\textsc{Redex}\ (r_1,x)) = \bot$ *for all* r_1.

Proof. A direct consequence of the implementation. $\texttt{check}_\mathfrak{s}$ never examines the route r. □

Lemma 4.5.14. *If* $\mathsf{wf}_\mathfrak{s}(c, m)$ *and* $\mathsf{pre}_\mathfrak{s}$, *then if*

$$\texttt{redexList}_\mathfrak{s}\ g\ c\ m\ !!\ i = \textsc{Redex}\ (r, (c_1, \textsc{Action}\ a))$$

then $c_1 \sqsubseteq_\mathfrak{s} c$ – *that is,* $\texttt{ap}\ c_1\ a = \textsc{True}$ *implies* $\texttt{ap}\ c\ a = \textsc{True}$.

Proof. Induction on m. For base terms, only $\texttt{action}\ a$ meets the pre-condition and $\texttt{ap}\ c\ a = \textsc{True}$ implies $\texttt{ap}\ c\ a = \textsc{True}$. If $m = m_1\ \texttt{>>=}\ f$ then apply Lemma 4.4.2 and IH. If $m = m_l\ |||^p_*\ m_r$ then apply either Lemma 4.5.7 or Lemma 4.5.8 depending on the route r. If $r = (L{:}r')$ we can prove that $\texttt{redexTree}_\mathfrak{s}\ g'\ c_l\ m_l\ !!\ i_l = \textsc{Redex}\ (r', (c_1, \textsc{Action}\ a))$ for some i_l. With IH, prove $\texttt{ap}\ c_1\ a = \textsc{True}$ implies $\texttt{ap}\ c_l\ a = \textsc{True}$ and with $\mathsf{pre}_\mathfrak{s}$ prove that $c_l \sqsubseteq_\mathfrak{s} c$ and therefore $\texttt{ap}\ c\ a = \textsc{True}$. The proof for the right-hand side is similar. □

Lemma 4.5.15. *If* $\mathsf{pre}_\mathfrak{s}$, $c_l \lozenge_\mathfrak{s} c_r$, $\mathsf{wf}_\mathfrak{s}(c_r, m_r)$ *and the following hold*

$$\texttt{redexTree}_\mathfrak{s}\ g_r\ c_r\ m_r\ !!\ i = \textsc{Redex}\ (r_r, (c_{r1}, \textsc{Action}\ a_r))$$
$$\texttt{check}_\mathfrak{s}\ w\ (\textsc{Redex}\ (r_r, (c_{r1}, \textsc{Action}\ a_r))) = \bot$$
$$w \Vdash m_l \longrightarrow^{c_l} w_l \Vdash m_l'$$

then $\texttt{check}_\mathfrak{s}\ w_l\ (\textsc{Redex}\ (r_r, (c_{r1}, \textsc{Action}\ a_r))) = \bot$.

Proof. From Lemma 4.5.12, either $\texttt{ap}\ c_{r1}\ a_r \neq \textsc{True}$ or $\texttt{ap}\ c_{r1}\ a_r = \textsc{True} \wedge \texttt{wa}\ a_r\ w = \bot$. If it is the former then this will be unaffected by a reduced action, so regardless of w_l, $\texttt{check}_\mathfrak{s}$ will still fail. If $\texttt{ap}\ c_{r1}\ a_r = \textsc{True}$ and $\texttt{wa}\ a_r\ w = \bot$, then apply Lemma 4.5.14 to prove $\texttt{ap}\ c\ a_r = \textsc{True}$. Next examine the reduction of m_l. If it is silent, then $w_l = w$, so m_r can fail in the same way. If reducing m_l performs an action a_l then apply Lemma 4.3.4 to prove $\texttt{wa}\ a_l\ w = \textsc{False}$ and, from Lemma 4.3.6, $\texttt{ap}\ c_l\ a = \textsc{True}$. With $c_l \lozenge_\mathfrak{s} c_r$ we know $\mathsf{ally}_\mathfrak{s}(a_l, a_r)$, which guarantees that $\texttt{wa}\ a_r\ w_l = \bot$. This, by Lemma 4.5.12, proves that $\texttt{check}_\mathfrak{s}$ will also fail with world w_l. □

Lemma 4.5.16. *If* $\mathsf{wf}_\mathfrak{s}(c, m)$, $\mathsf{PRE}_\mathfrak{s}$, $\texttt{redexTree}_\mathfrak{s}\ g_1\ c\ m\ !!\ i = \textsc{Redex}\ (r_1, (c_1, \textsc{Action}\ a))$, $\texttt{check}_\mathfrak{s}\ w\ (\textsc{Redex}\ (r_1, (c_1, \textsc{Action}\ a))) = \bot$ *and* $w \Vdash m \xrightarrow{g_2 \mapsto r_2}_c w' \Vdash m'$ *then* $\texttt{check}_\mathfrak{s}\ w'\ (\textsc{Redex}\ (r_1, (c_1, \textsc{Action}\ a))) = \bot$

Proof. Induction on m.

- True by contradiction for all base redexes, since, being deterministic, they cannot both reduce and fail to reduce.

- For $m = m_1$ `>>=` f (m_1 not a value), from Lemma 4.4.2 $\texttt{redexTree}_\mathfrak{s}\ g_1\ c\ (m_1 \texttt{>>=}\ f) = \texttt{redexTree}_\mathfrak{s}\ g_1\ c\ m_1$ and from the language semantics it is possible to derive both $w \Vdash m_1 \xrightarrow{g_2 \mapsto r_2}{}_c\ w' \Vdash m_1'$ and $m' = m_1' \texttt{>>=}\ f$ from $w \Vdash m_1 \texttt{>>=}\ f \xrightarrow{g_2 \mapsto r_2}{}_c\ w' \Vdash m'$. Then just apply IH.

- When $m = m_l\ \texttt{|||}^p_*\ m_r$, where m_l and m_r are not both values, perform case analysis on the routes r_1 and r_2. Neither can be [] (Lemma 4.5.9, Lemma 4.2.5).

  - If both redexes are on the same side, say the left-hand side, making $r_1 = (L{:}r_1')$ and $r_2 = (L{:}r_2')$, apply Lemma 4.5.7 and Lemma 4.2.10. This lets one prove with IH that $\texttt{check}_\mathfrak{s}\ w'$ (Redex $(r_1', (c_1, \text{Action } a))) = \bot$. Then apply Lemma 4.5.13 to prove the same is true with route $(L{:}r_1')$.
  - If the redexes are on opposite sides then from $\texttt{pf}\ p\ c = (c_l, c_r)$ and $\mathsf{PRE}_\mathfrak{s}$ we can prove $\mathsf{pre}_\mathfrak{s}$ and $c_l \Diamond_\mathfrak{s} c_r$ (and $c_r \Diamond_\mathfrak{s} c_l$ by symmetry). Apply Lemma 4.5.15.

□

We are finally in a position to "repackage" divergence. Lemma 4.5.17 shows that re-using a previous route as a Guess will mean the redex at that route will be the first to be chosen (a re-working of the proof of both Lemma 4.3.7 and Lemma 4.3.9). Lemma 4.5.18 uses this to tidy up Lemma 4.5.16, and Lemma 4.5.19 follows exactly the same procedure as Lemma 4.5.6 to prove failure implies divergence.

Lemma 4.5.17. *If* $\mathsf{wf}_\mathfrak{s}(c, m)$ *and* $\texttt{redexList}_\mathfrak{s}\ g\ c\ m\ \texttt{!!}\ i = \text{Redex } (r, x)$ *then we know that* $\texttt{redexList}_\mathfrak{s}\ r\ c\ m\ \texttt{!!}\ 0 = \text{Redex } (r, x)$.

Proof. Induction on m. For base terms, $\bot$ is not well-formed and otherwise there is just one potential redex, the Guess is ignored and i must be 0 already. If $m = m_1$ `>>=` f where m_1 is not a value, use Lemma 4.4.2 and IH. If $m = m_l\ \texttt{|||}^p_*\ m_r$, where m_l and m_r are not both values, then do case analysis on r. If $r = (L{:}r')$, apply Lemma 4.5.7 to prove that for some i_l, $\texttt{redexList}_\mathfrak{s}\ g'\ c_l\ m_l\ \texttt{!!}\ i_l = \text{Redex } (r', x)$, where $g = (b{:}g')$. By IH, this implies $\texttt{redexList}_\mathfrak{s}\ r'\ c_l\ m_l\ \texttt{!!}\ 0 = \text{Redex } (r', x)$, and by Lemma 4.4.3 this can be shown to imply $\texttt{redexList}_\mathfrak{s}\ (L{:}r')\ c\ (m_l\ \texttt{|||}^p_*\ m_r)\ \texttt{!!}\ 0 = \text{Redex } ((L{:}r'), x)$. That is: if x is the first potential redex in m_l with guess r' then it will be the first potential redex in $m_l\ \texttt{|||}^p_*\ m_r$ with guess $(L{:}r')$, since it will be m_l that is searched first. If $r = (R{:}r')$ then the proof is similar. □

Lemma 4.5.18. *If* $\mathsf{PRE}_\mathfrak{s}$, $\mathsf{wf}_\mathfrak{s}(c, m)$, $w \Vdash m\ {}^{g \mapsto \bot}\!\uparrow^c$ *and* $w \Vdash m \longrightarrow^c w' \Vdash m'$, *then for some* g_1 $w' \Vdash m'\ {}^{g_1 \mapsto \bot}\!\uparrow^c$.

Proof. First apply Lemma 4.5.11 to prove that for some i $\texttt{redexList}_\mathfrak{s}\ g\ c\ m\ \texttt{!!}\ i =$ Redex $(r, (c_1, \text{Action } a))$ and $\texttt{check}_\mathfrak{s}\ w$ Redex $(r, (c_1, \text{Action } a)) = \bot$. Using Lemma 4.5.16 we can then prove $\texttt{check}_\mathfrak{s}\ w'$ Redex $(r, (c_1, \text{Action } a)) = \bot$ (a can

still cause failure in world w'). Apply Lemma 4.5.17 to prove $\texttt{redexList}_{\mathfrak{s}}\ r\ c\ m\ !!\ 0 = \text{REDEX}\ (r, (c_1, \text{ACTION}\ a))$ – if we chose $\mathsf{Guess}\ r$ the second time then the first redex to be chosen will be action a. It can be shown easily that this means $\texttt{nextR}_{\mathfrak{s}}\ r\ c\ w'\ m' = \bot$, and therefore, letting $g_1 = r$, $w' \Vdash m' \overset{g_1 \mapsto \bot}{\uparrow^c}$. □

Lemma 4.5.19. *(Part 3 of Theorem 4.5.1). If* $\mathsf{PRE}_{\mathfrak{s}}$*, then if* $\mathsf{wf}_{\mathfrak{s}}(c, m)$ *and* $w \Vdash m \overset{g \mapsto \bot}{\uparrow^c}$*, then* $w \Vdash m \Uparrow^c$.

Proof. Induction on the number of reduction steps it might take to reach normal form. Base case (0): since $w \Vdash m \uparrow^c$, by Lemma 4.3.2 it can't be in normal form. Inductive case: it can't converge to normal form after $i+1$ steps because, by Lemma 4.5.18, after one step it can still fail, and by IH it can't converge to normal form after i steps. □

4.6 Confluence of reduction

The confluence result is finally proved in this section. As is standard for confluence proofs, this is achieved by proving a so-called "diamond property". This is a proof that if two different non-deterministic reductions are possible, then there exists a common reduct to which both sides can then reduce.

We assume for all of this sub-section that $\mathsf{PRE}_{\mathfrak{s}}$ holds.

Lemma 4.6.1. *If* $w \Vdash m \overset{1}{\downarrow\!\!\downarrow^c} w' \Vdash m'$ *(i.e.* $w \Vdash m \longrightarrow^c w' \Vdash m'$ *and* $w' \Vdash m' \downarrow^c$*), then* $w \Vdash m \overset{1}{\Downarrow^c} w' \Vdash m'$*. (This is a proof of confluence for programs which require one reduction step).*

Proof. Induction on m.

- All base programs either cannot reduce or reduce deterministically.
- With m_1 `>>=` f (m_1 not a value), we know that $w \Vdash m_1 \longrightarrow^c w' \Vdash m_1'$, $w' \Vdash m_1' \downarrow^c$ and $m' = m_1'$ `>>=` f. Because m' is stalled, m_1' cannot be a value, so apply IH.
- With m_l `|||`$^p_*$ m_r first prove that neither side can fail (if one side did fail, then the whole program could fail, and, by Theorem 4.5.1, the whole program could never reduce to a normal form – but it does, after one step). Next, prove that it's impossible for *both* sides to be successfully reduced (if both were, then by Theorem 4.3.1, after single-step reducing one particular side the program couldn't be in normal form – the opposite side could still either be reduced or fail). So any two reductions must be both in m_l or both in m_r. Apply IH to prove that both reductions on that side will have the same outcome.

□

Lemma 4.6.2. *Diamond Property. If* $w \Vdash m \overset{g' \mapsto r'}{\longrightarrow_c} w' \Vdash m'$ *and* $w \Vdash m \overset{g'' \mapsto r''}{\longrightarrow_c} w'' \Vdash m''$ *then either*

- *The same redex was reduced:* $w' = w''$ *and* $m' = m''$.
- *There is a common reduct: for some* w''' *and* m''', $w' \Vdash m' \xrightarrow{r'' \mapsto r''}_c w''' \Vdash m'''$ *and* $w'' \Vdash m'' \xrightarrow{r' \mapsto r'}_c w''' \Vdash m'''$.
- *Both sides will diverge:* $w' \Vdash m' \Uparrow^c$ *and* $w'' \Vdash m'' \Uparrow^c$.

Actually, in the last case we prove something slightly more specific which always implies divergence. This is because actual divergence is inadmissible, requiring existential quantification and negation. We prove instead that they either fail or reduce to a badly-formed program. Both of these imply divergence, and it is easy to prove that if a "`>>=`" or a "`par`" diverge in this special way then the whole program will also diverge in this way.

Proof. Induction on m.

- All base reduction rules either can't reduce at all (contradiction), or deterministically reduce to the same redex.
- $w \Vdash m_1$ `>>=` $f \longrightarrow^c w' \Vdash m_1'$ `>>=` f and $w \Vdash m_1$ `>>=` $f \longrightarrow^c w'' \Vdash m_1''$ `>>=` f. If m_1' or m_1'' are stalled, then by Lemma 4.6.1, $w' = w''$ and $m_1' = m_1''$. This case rules out either m_1' or m_1'' being values, so the next reduction must be within m_1' and m_1'', and not the application of f to something. Apply IH.
- $w \Vdash m_l$ `|||`$^p_*$ $m_r \longrightarrow^c w' \Vdash m_l'$ `|||`$^p_*$ m_r' and $w \Vdash m_l$ `|||`$^p_*$ $m_r \longrightarrow^c w'' \Vdash m_l''$ `|||`$^p_*$ m_r''. The two reductions may have reduced different sides, or the same side.
  - If the same side was reduced, say that of m_l, use Lemma 4.6.1 to account for the possibility of m_l' or m_l'' being stalled or values (which would cause reduction to flip to the other side the second time around). If either side was stalled or a value then since it was the last reduction step before convergence to normal form both reductions would have been the same. We can now apply IH.
  - If one reduction was in m_l and the other was in m_r, then apply Theorem 4.3.1. Either both reduction orders can then fail (which entails failure for the whole program and therefore divergence by Theorem 4.5.1), or both reduction orders can reduce once more to get the same result by reducing the opposite side to that initially reduced. However, if one side reduces to a program which is $\bot$ (which is still a successful reduction), then depending on the ordering, the parallel execution may accidentally force the program's evaluation.

    This is why we need to mention divergence (as opposed to failure) in the diamond property, and is a consequence of the side condition that $m_l \neq \bot$ and $m_r \neq \bot$ in the single-step semantics. As an example, consider the program

    ```
    par LeftWr (action (WriteI 6)) (return 2 >>= \_ -> ⊥) f
    ```

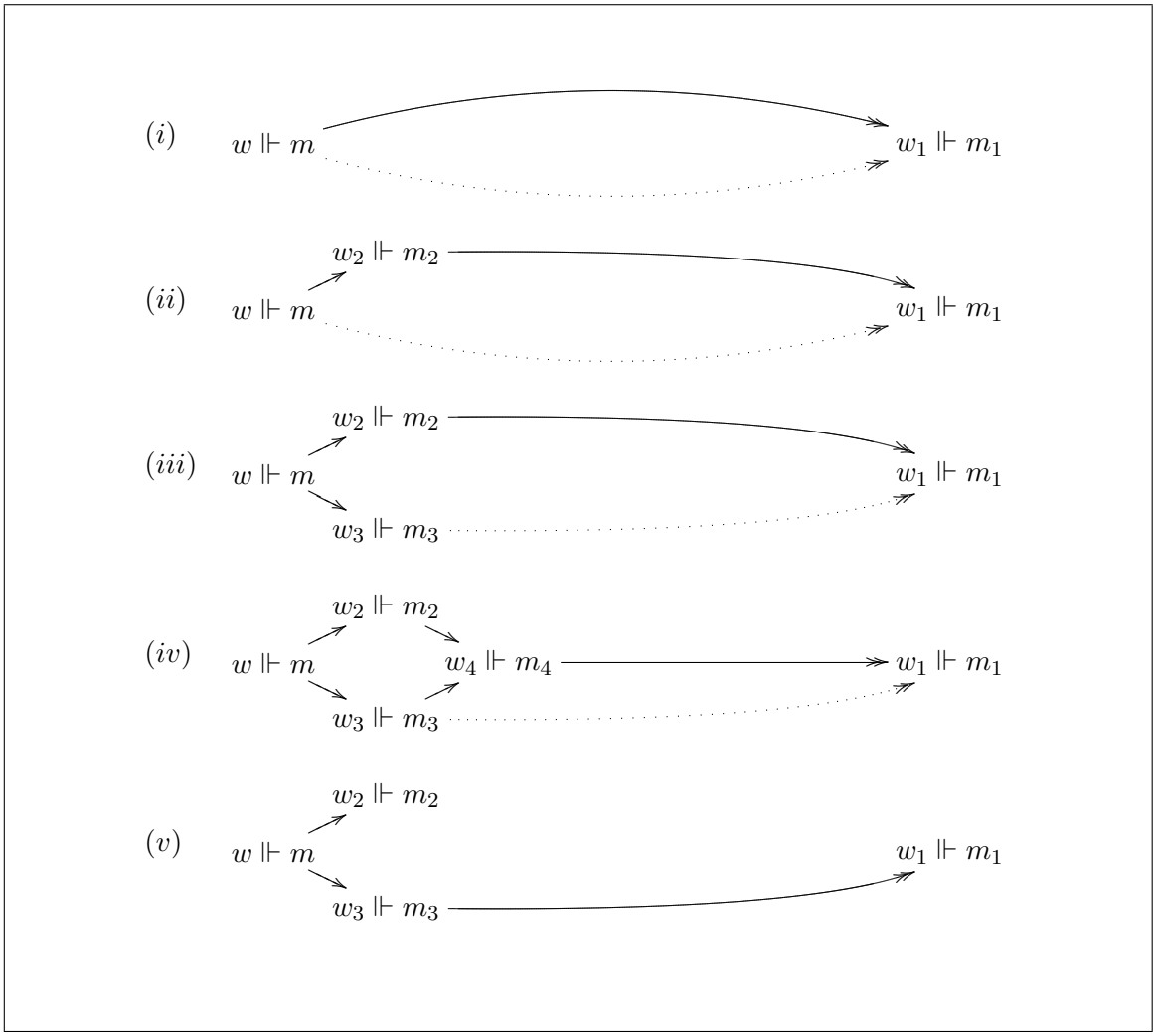


Figure 4.5: Proving confluence using the diamond property

The left-hand side reduces to (`return` 0) and the right-hand side reduces to $\bot$. Reducing the left side then the right side yields `par LeftWr (return 0)` $\bot$ `f`, a badly-formed program (which diverges), but reducing the right side and then the left will fail outright (which causes divergence).

□

Theorem 4.6.1. *Confluence. If* $w \Vdash m \; {}^{i}\!\downarrow\!\!\!\downarrow^{c} \; w_1 \Vdash m_1$ *then* $w \Vdash m \; {}^{i}\!\Downarrow^{c} \; w_1 \Vdash m_1$.

Proof. Induction on i. If $i = 0$, then apply Lemma 4.3.2. In the inductive case ($i = k + 1$), first prove for $k = 0$ using Lemma 4.6.1. Figure 4.5 demonstrates the sequence of steps required to prove the full inductive case. Diagram (i) is the initial proof obligation.

Assuming $k \geq 1$ (and $i > 1$), extract the first of the $k + 1$ reduction steps to yield $w \Vdash m \longrightarrow^{c} w_2 \Vdash m_2$ and $w_2 \Vdash m_2 \; {}^{k}\!\downarrow\!\!\!\downarrow^{c} \; w_1 \Vdash m_1$, for some w_2, m_2 (diagram (ii)). To prove confluence, we must show that for any arbitrary reduction of $w \Vdash m$, it will eventually reduce

to $w_1 \Vdash m_1$. First prove that despite the non-determinism, $w \Vdash m$ cannot fail or be in normal form. (It can't be in normal form because it can reduce – Lemma 4.3.2; it can't fail because it does eventually converge to normal form – Theorem 4.5.1). Therefore, the only alternative to reducing to $w_2 \Vdash m_2$ is that there is some other $w_3 \Vdash m_3$ such that $w \Vdash m \longrightarrow^c w_3 \Vdash m_3$ (diagram (*iii*)). We must now show that $w_3 \Vdash m_3 \ {}^{k}\Downarrow^{c} \ w_1 \Vdash m_1$. Apply IH twice in a row: first in a forwards style to prove that we know $w_2 \Vdash m_2 \ {}^{k}\Downarrow^{c} \ w_1 \Vdash m_1$; second in a backwards style to show that we need only prove $w_3 \Vdash m_3 \ {}^{k}\downarrow^{c} w_1 \Vdash m_1$. Now, since we know that $w \Vdash m \longrightarrow^c w_2 \Vdash m_2$ and $w \Vdash m \longrightarrow^c w_3 \Vdash m_3$, from the diamond property (Lemma 4.6.2), we can prove that either:

- $w_2 = w_3$ and $m_2 = m_3$: since we know that $w_2 \Vdash m_2 \ {}^{k}\downarrow^{c} w_1 \Vdash m_1$, it is trivially true that $w_3 \Vdash m_3 \ {}^{k}\downarrow^{c} w_1 \Vdash m_1$.
- There is a common reduct $w_4 \Vdash m_4$ (diagram (*iv*) in Figure 4.5) such that $w_2 \Vdash m_2 \longrightarrow^c w_4 \Vdash m_4$ and $w_3 \Vdash m_3 \longrightarrow^c w_4 \Vdash m_4$: from the confluence of reduction of $w_2 \Vdash m_2$, prove that $w_4 \Vdash m_4 \ {}^{k-1}\downarrow^{c} \ w_1 \Vdash m_1$. Thus $w_3 \Vdash m_3 \ {}^{k}\downarrow^{c} \ w_1 \Vdash m_1$ (diagram (*v*) in Figure 4.5).
- $w_2 \Vdash m_2 \Uparrow^c$ and $w_3 \Vdash m_3 \Uparrow^c$: proof by contradiction – we know that $w_2 \Vdash m_2$ can converge to normal form.

□

Corollary 4.6.3. *Either $w \Vdash m \Uparrow^c$ or there exists a w', m' such that $w \Vdash m \Downarrow^c w' \Vdash m'$ (and not both).*

Proof. The definition $w \Vdash m \Uparrow^c$ is that it's not the case that $w \Vdash m \downarrow^c w' \Vdash m'$, and $w \Vdash m \downarrow^c w' \Vdash m'$, by confluence, is equivalent to $w \Vdash m \Downarrow^c w' \Vdash m'$. □

4.7 Chapter summary

This chapter presented the full, machine-verified proof that $\mathsf{PRE}_{\mathfrak{s}}$ guarantees confluence. This required a great many technicalities, including two re-implementations of single-step reduction, a theorem which related failure to divergence, a notion of "well-formedness" of Curio programs, and, as is usual for confluence proofs, a diamond property. Although many of the technicalities are of no relevance to the rest of the dissertation, the result in itself is the most important single theorem in this document.

Chapter 5 takes the first step towards the construction of a large, practical I/O model by developing "combinators". These are functions which modify and combine existing I/O models in a way which preserves pre-condition $\mathsf{PRE}_{\mathfrak{s}}$.

Chapter 5

A toolkit for building I/O models

Up until now, we have had to prove the precondition $\mathsf{PRE}_{\mathtt{s}}$ on a case-by-case basis for each individual I/O model. The proofs all required a little creativity and effort, and this becomes problematic for larger models. What we want is the ability to combine confluent I/O models together in a way which both preserves confluence and has a useful common-sense intuition.

In this chapter we investigate some powerful tools that help in the construction of confluent I/O models. These are:

- Some "combinators" which allow confluent I/O models to be modified and combined in such a way as to preserve confluence.
- A notion of "location-based" I/O model. These models have a close formal relationship to normal I/O models and encode the idea that certain actions need to be able to distinguish different calling processes.

These tools and formalisms are intended mainly as a stepping-stone towards modelling a real API. This is the goal of Chapter 6. In an ideal world one would like there to be a single correct I/O model which perfectly captures the reasonable expected behaviour of the real API. If this was the case we might even omit combinators completely and proceed towards trying to model a real API, solving each technical program as it arises.

However, in reality what constitutes a correct I/O model is open to debate. Therefore, we feel that it is better, initially, to tackle the modelling of APIs in as general a way as possible. By constructing general purpose tools one gets a stronger insight into the flexibility of Curio and one also confronts head-on the specific problems with modelling an API – for example, dynamic allocation. Furthermore, it was found that without the use of combinators the construction and machine-verification of complex I/O models became unwieldy. The main danger is that the details in this chapter may give the false impression that a programmer will be expected to deal with many different APIs, large and small, when writing programs in Curio. We give small example programs for many different I/O models, but these are primarily just to help the reader.

$$(*) \;::\; \mathsf{IOModel}\ \nu_1\ \alpha_1\ \rho_1\ \omega_1\ \varsigma_1 \to \mathsf{IOModel}\ \nu_2\ \alpha_2\ \rho_2\ \omega_2\ \varsigma_2 \to \mathsf{IOModel}\ (\mathsf{Either}\ \nu_1\ \nu_2)\ (\mathsf{Either}\ \alpha_1\ \alpha_2)\ (\rho_1,\rho_2)\ (\omega_1,\omega_2)\ (\varsigma_1,\varsigma_2)$$

$$\langle \mathtt{af}_1, \mathtt{wa}_1, \mathtt{ap}_1, \mathtt{pf}_1\rangle * \langle \mathtt{af}_2, \mathtt{wa}_2, \mathtt{ap}_2, \mathtt{pf}_2\rangle \triangleq \langle \mathtt{af}_1 * \mathtt{af}_2, \mathtt{wa}_1 * \mathtt{wa}_2, \mathtt{ap}_1 * \mathtt{ap}_2, \mathtt{pf}_1 * \mathtt{pf}_2\rangle$$

$$\mathtt{af}_1 * \mathtt{af}_2 \triangleq \lambda a.\ \lambda(w_1,w_2).\ \begin{cases} ((w_1',w_2),\textsc{Left}\ v_1), & a = \textsc{Left}\ a_1, \mathtt{af}_1\ a_1\ w_1 = (w_1',v_1) \\ ((w_1,w_2'),\textsc{Right}\ v_2), & a = \textsc{Right}\ a_2, \mathtt{af}_2\ a_2\ w_2 = (w_2',v_2) \end{cases}$$

$$\mathtt{wa}_1 * \mathtt{wa}_2 \triangleq \lambda a.\ \lambda(w_1,w_2).\ \begin{cases} \mathtt{wa}_1\ a_1\ w_1, & a = \textsc{Left}\ a_1 \\ \mathtt{wa}_2\ a_2\ w_2, & a = \textsc{Right}\ a_2 \end{cases}$$

$$\mathtt{ap}_1 * \mathtt{ap}_2 \triangleq \lambda(c_1,c_2).\ \lambda a.\ \begin{cases} \mathtt{ap}_1\ c_1\ a_1, & a = \textsc{Left}\ a_1 \\ \mathtt{ap}_2\ c_2\ a_2, & a = \textsc{Right}\ a_2 \end{cases}$$

$$\mathtt{pf}_1 * \mathtt{pf}_2 \triangleq \lambda(p_1,p_2).\ \lambda(c_1,c_2).\ ((c_{1l},c_{2l}),(c_{1r},c_{2r})), \quad \text{where } \mathtt{pf}_1\ p_1\ c_1 = (c_{1l},c_{1r}),\quad \mathtt{pf}_2\ p_2\ c_2 = (c_{2l},c_{2r})$$

Figure 5.1: The cartesian product of I/O models

In this chapter we make extensive use of strictness annotation in our proofs and data-structures, but all details of this will be omitted. Effectively every structure that we utilise is strict. This means that problems associated with reasoning about infinite or partial structures, such as admissibility, are not present. Also, a great many smaller results contain definedness side-conditions and these are completely removed to make the presentation clearer. All important lemmas or theorems are presented as is, however. In particular, no combinator requires the proof of obscure, hidden side-conditions about the I/O models that one is manipulating. All omitted definitions can be found in Section A.2.

We begin in Section 5.1 by describing two quite simple combinators for traditional I/O models. Section 5.2 then addresses the problem of dynamic allocation in Curio, and presents a combinator called `dmap`, which is a basic, flawed attempt at a solution. Section 5.3 motivates the need for a new type of I/O model, introduces "location-based" I/O models, and demonstrates their close formal relationship to normal I/O models. Finally, Section 5.4 introduces `dmap'`, a location-based version of the `dmap` combinator which has more desirable properties.

5.1 Simple combinators

An I/O model is just a data-structure in our metalanguage and a combinator is a function which creates a new I/O model data-structure from one or more existing ones in such a way that confluence is fully preserved. We now present two useful combinators.

5.1.1 Cartesian product combinator

Figure 5.1 describes the cartesian product combinator '$*$'. $\mathfrak{s}_1 * \mathfrak{s}_2$ represents the "gluing together" of I/O models $\mathfrak{s}_1$ and $\mathfrak{s}_2$. The world state becomes the cartesian product of the world states of $\mathfrak{s}_1$ and $\mathfrak{s}_2$, and an action can be either an action from $\mathfrak{s}_1$ or one from $\mathfrak{s}_2$. Similarly, a context must be able to determine whether actions from either $\mathfrak{s}_1$ or $\mathfrak{s}_2$ are allowed and, when performing concurrency, one must specify the permissions given to each sub-program for *both* sides of the world state.

Theorem 5.1.1. *If* $\mathsf{PRE}_{\mathfrak{s}_1}$ *and* $\mathsf{PRE}_{\mathfrak{s}_2}$ *then* $\mathsf{PRE}_{\mathfrak{s}_1 * \mathfrak{s}_2}$.

Proof. When splitting contexts, if $(\mathtt{pf}_1 * \mathtt{pf}_2)\ (p_1, p_2)\ (c_1, c_2) = ((c_{1l}, c_{2l}), (c_{1r}, c_{2r}))$, then that means $\mathtt{pf}_1\ p_1\ c_1 = (c_{1l}, c_{1r})$ and $\mathtt{pf}_2\ p_2\ c_2 = (c_{2l}, c_{2r})$. The proof proceeds as follows:

- $c_{1l} \sqsubseteq_{\mathfrak{s}_1} c_1$ and $c_{2l} \sqsubseteq_{\mathfrak{s}_2} c_2$ imply $(c_{1l}, c_{2l}) \sqsubseteq_{\mathfrak{s}_1 * \mathfrak{s}_2} (c_1, c_2)$. If the action permitted by context (c_{1l}, c_{2l}) is from $\mathfrak{s}_1$ then use $c_{1l} \sqsubseteq_{\mathfrak{s}_1} c_1$ to show that it is permitted by context (c_1, c_2). Otherwise use $c_{2l} \sqsubseteq_{\mathfrak{s}_2} c_2$.
- $c_{1r} \sqsubseteq_{\mathfrak{s}_1} c_1$ and $c_{2r} \sqsubseteq_{\mathfrak{s}_2} c_2$ imply $(c_{1r}, c_{2r}) \sqsubseteq_{\mathfrak{s}_1 * \mathfrak{s}_2} (c_1, c_2)$. Similar to above.
- $c_{1l} \diamondsuit_{\mathfrak{s}_1} c_{1r}$ and $c_{2l} \diamondsuit_{\mathfrak{s}_2} c_{2r}$ imply $(c_{1l}, c_{2l}) \diamondsuit_{\mathfrak{s}_1 * \mathfrak{s}_2} (c_{1r}, c_{2r})$: either the permitted actions modify opposite sides of the tuple or the same side.
  - Different sides: $\textsc{Left}\ a_1 \mid\mid\mid_{\mathfrak{s}_1 * \mathfrak{s}_2} \textsc{Right}\ a_2$, $\mathsf{ally}_{\mathfrak{s}_1 * \mathfrak{s}_2}(\textsc{Left}\ a_1, \textsc{Right}\ a_2)$ and $\mathsf{ally}_{\mathfrak{s}_1 * \mathfrak{s}_2}(\textsc{Right}\ a_2, \textsc{Left}\ a_1)$ are true because neither action modifies the side of world state that affects the other.
  - Same sides: $\textsc{Right}\ a_2 \mid\mid\mid_{\mathfrak{s}_1 * \mathfrak{s}_2} \textsc{Right}\ a_2'$, $\mathsf{ally}_{\mathfrak{s}_1 * \mathfrak{s}_2}(\textsc{Right}\ a_2, \textsc{Right}\ a_2')$ and $\mathsf{ally}_{\mathfrak{s}_1 * \mathfrak{s}_2}(\textsc{Right}\ a_2', \textsc{Right}\ a_2)$ are true because $c_{2l} \diamondsuit_{\mathfrak{s}_2} c_{2r}$, $\mathtt{ap}_2\ c_{2l}\ a_2 = \textsc{True}$ and $\mathtt{ap}_2\ c_{2r}\ a_2' = \textsc{True}$. (The proof is similar if both actions affect the left-hand side world state).

□

For example, the $\mathfrak{lock} * \mathfrak{bfft}$ I/O model has the functionality of both $\mathfrak{lock}$ and $\mathfrak{bfft}$. The sample program in Figure 5.2 gives a quick (if mostly useless) example of how one programs with this sort of I/O model. The next subsection contains a more practical example.

5.1.2 String map combinator

The second combinator is a string map. This turns an I/O model $\mathfrak{s}$ into one in which the world state is a map from String to the world state of $\mathfrak{s}$. The `smap` combinator has the following type.

$$\begin{aligned} \mathsf{SMap}\ \beta &\triangleq \mathsf{String} \to \beta \\ \mathtt{smap} &:: \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \to \\ &\qquad \mathsf{IOModel}\ \nu\ (\mathsf{String}, \alpha)\ [(\mathsf{String}, \mathsf{Splitter}\ \rho)]\ (\mathsf{SMap}\ \omega)\ (\mathsf{SMap}\ (\mathsf{Cxt}\ \varsigma)) \end{aligned}$$

```
sample :: Prog_{lock*bfft} Int
sample = do
  action (Left Lock)                    -- lock the mutex
  par                                   -- run two concurrent processes
    ((),True)                           -- give send permission to LHS
    (action (Right (Send 13)))          -- LHS: send 13
    (do
       Right i1 <- action (Right Rcve)  -- RHS: receive the integer i1
       action (Left Unlock)             --      unlock the mutex
       return i1)                       --      return the integer
    (\x y -> y)                         -- return the RHS's return value
```

Figure 5.2: Sample cartesian product program

This is similar to the cartesian product, except there are an infinite number of $\mathfrak{s}$ models, each indexed by a different String (String could just as easily be replaced with any other type with an equality function defined on it, but this generality does not interest us). This is a useful model of indexed global storage, such as a file system. An action (s,a) performs action a from $\mathfrak{s}$ in the world state associated with name s.

The mapping is implemented using an actual function since this is simple to implement and obeys the best extensional properties. As a data-structure it is extremely inefficient but this is of no concern to us, since we are only interested in its properties as a formal model. These maps are manipulated with the following operations:

$$\begin{aligned}
\texttt{lkpM} &:: \forall\beta.\,\forall\gamma.\,\texttt{Eq}\;\gamma \Rightarrow \gamma \to (\gamma \to \beta) \to \beta \\
\texttt{ovwM} &:: \forall\beta.\,\forall\gamma.\,\texttt{Eq}\;\gamma \Rightarrow \gamma \to \beta \to (\gamma \to \beta) \to (\gamma \to \beta) \\
\texttt{maskM} &:: \forall\beta.\,\forall\gamma.\,\texttt{Eq}\;\gamma \Rightarrow (\gamma \to \mathsf{Bool}) \to \beta \to (\gamma \to \beta) \to (\gamma \to \beta) \\
\texttt{stM} &:: \forall\beta.\,\forall\gamma.\,\forall\delta.\,\texttt{Eq}\;\gamma \Rightarrow (\beta \to (\beta,\delta)) \to \gamma \to (\gamma \to \beta) \to (\gamma \to \beta,\delta)
\end{aligned}$$

`lkpM` s m queries the value indexed by s by applying the map function. `ovwM` s v m overwrites the existing map by replacing the value pointed to by s with v. `maskM` f v m modifies map m such that every $s :: \gamma$ for which $f\ s =$ TRUE is modified to point to v. `stM` f s is defined using `lkpM` and `ovwM`, and creates a new state-transformer on an entire map by using a state-transformer f on the item indexed by s, and returning that state-transformer's value.

The above types contain "Eq γ", which suggests that we use Haskell's type-class mechanism to overload the operator `(==)` $:: \gamma \to \gamma \to$ Bool, guaranteeing that we can compute the equality of two terms of type γ. We don't do this, but the idea is the same. Sparkle currently has no facilities for ad-hoc polymorphism*, so we instead just have a few duplicates, one for each specific type γ which we wish to use.

*This is current research. See [119].

Lemma 5.1.1. *For each particular type γ used, the operations* `lkpM`*,* `ovwM`*,* `maskM` *and* `stM` *obey the following properties*[†]*:*

1. *If* $s_1 \neq s_2$ *then* $\texttt{lkpM}\, s_1\ (\texttt{ovwM}\, s_2\ v\ m) = \texttt{lkpM}\, s_1\ m$.
2. $\texttt{lkpM}\, s\ (\texttt{ovwM}\, s\ v\ m) = v$
3. $\texttt{ovwM}\, s\ v_1\ (\texttt{ovwM}\, s\ v_2\ m) = \texttt{ovwM}\, s\ v_1\ m$
4. *If* $s_1 \neq s_2$ *then* $\texttt{ovwM}\, s_1\ v_1\ (\texttt{ovwM}\, s_2\ v_2\ m) = \texttt{ovwM}\, s_2\ v_2\ (\texttt{ovwM}\, s_1\ v_1\ m)$
5. *If* $f\ s = \textsc{True}$ *then* $\texttt{lkpM}\, s\ (\texttt{maskM}\, f\ v\ m) = v$.
6. *If* $f\ s = \textsc{False}$ *then* $\texttt{lkpM}\, s\ (\texttt{maskM}\, f\ v\ m) = \texttt{lkpM}\, s\ m$.
7. *If* $s_1 \neq s_2$ *then* $\mathsf{cmmt}(\texttt{stM}\, f_1\ s_1, \texttt{stM}\, f_2\ s_2, w)$.
8. *If* $\mathsf{cmmt}(f_1, f_2, \texttt{lkpM}\, s\ w)$ *then* $\mathsf{cmmt}(\texttt{stM}\, f_1\ s, \texttt{stM}\, f_2\ s, w)$

Proof. These are all proved without difficulty using case analysis, extensionality and basic properties of `==`. □

When performing concurrency, one must supply a list of how the permissions for each String are to be distributed between the left- and right-hand sides. If a String n isn't mentioned in the list then all the current permissions will be given to the right side, preventing the left side from doing anything at all with the data indexed by n.

How do we give no permissions at all to a particular side? The type constructors Cxt and Splitter are used for this purpose and are defined as follows:

$$\begin{aligned} \mathsf{Cxt}\ \varsigma &\triangleq \textsc{CxtB} \mid \textsc{Cxt}\ \varsigma\ \mathsf{Bool} \\ \mathsf{Splitter}\ \rho &\triangleq \textsc{AllLeft} \mid \textsc{Split}\ \rho \mid \textsc{AllRight} \end{aligned}$$

The type $\mathsf{Cxt}\ \varsigma$ denotes the context ς extended to guarantee the existence of a context which permits no actions and a context which permits every action. This is necessary because we want all permissions to go to one side by default. (In Chapter 8 we investigate a mathematical model of contexts which naturally incorporates both "all actions" and "no action" contexts).

The context CxtB forbids all actions, and any attempt to split CxtB will result in CxtB on both sides. A context that permits all actions is more subtle to model because it is unclear how one should split such a context. This is solved using a small hack. The context (Cxt c False) permits the same actions as context c and it will split into (Cxt c_l False) and (Cxt c_r False) as determined by some $p :: \rho$. The context (Cxt c True) permits all

[†]The definition of the cmmt macro can be found in Figure 2.1 on page 24. $\mathsf{cmmt}(f_1, f_2, w)$ is the formal definition of what it means for state-transformers f_1 and f_2 to be order independent on world state w.

actions, but c is still necessary since, when we split the context, the sub-contexts will then become (CXT c_l FALSE) and (CXT c_r FALSE).

The type Splitter is used to split contexts of type Cxt ς. SPLIT p splits the context as specified above using p. ALLLEFT gives all permissions to the left-hand side and ALLRIGHT gives all permissions to the right-hand side.

The functions `apCxt` and `pfCxt` implement this. The function `splitM` is used to split a context map into two separate maps. `splitM pf` f ps cm splits a context map cm of type $\gamma \to$ Cxt ς into two contexts of the same type as specified by the list ps, each of element of which is instructions on how the context of a particular γ is to be split. The function f describes the default behaviour when a γ is not mentioned in ps. If $f\ s =$ TRUE all permissions for s go to the left-hand side; if $f\ s =$ FALSE all permissions for s go to the right-hand side.

Their types are as follows:

$$\begin{aligned}
\texttt{apCxt} &:: \forall\varsigma.\, \forall\alpha.\, (\varsigma \to \alpha \to \mathsf{Bool}) \to \mathsf{Cxt}\ \varsigma \to \alpha \to \mathsf{Bool} \\
\texttt{pfCxt} &:: \forall\varsigma.\, \forall\rho.\, (\rho \to \varsigma \to (\varsigma,\varsigma)) \to \mathsf{Splitter}\ \rho \to \mathsf{Cxt}\ \varsigma \to (\mathsf{Cxt}\ \varsigma, \mathsf{Cxt}\ \varsigma) \\
\texttt{splitM} &:: \forall\rho.\, \forall\varsigma.\, \forall\gamma.\, \mathsf{Eq}\ \gamma \Rightarrow (\rho \to \varsigma \to (\varsigma,\varsigma)) \to (\gamma \to \mathsf{Bool}) \to [(\gamma, \mathsf{Splitter}\ \rho)] \to \\
&\qquad (\gamma \to \mathsf{Cxt}\ \varsigma) \to (\gamma \to \mathsf{Cxt}\ \varsigma, \gamma \to \mathsf{Cxt}\ \varsigma)
\end{aligned}$$

The definition for `smap` can now be given as follows:

$$\begin{aligned}
\texttt{smap}\ \langle \texttt{af}, \texttt{wa}, \texttt{ap}, \texttt{pf} \rangle \triangleq\ & \langle \lambda(s,a).\ \texttt{stM}\ (\texttt{af}\ \texttt{a})\ \texttt{s}, \\
& \lambda(s,a).\ \lambda m.\ \texttt{wa}\ a\ (\texttt{lkpM}\ s\ m), \\
& \lambda c.\ \lambda(s,a).\ \texttt{apCxt ap}\ (\texttt{lkpM}\ s\ c)\ a, \\
& \texttt{splitM pf}\ (\lambda c.\ \text{FALSE})\rangle
\end{aligned}$$

Lemma 5.1.2. *If* $\texttt{splitM pf}\ f\ ps\ c = (c_l, c_r)$ *then for all* $s :: \gamma$ *there exists some* $p :: \rho$ *such that* $\texttt{pfCxt pf}\ p\ (\texttt{lkpM}\ s\ c) = (\texttt{lkpM}\ s\ c_l, \texttt{lkpM}\ s\ c_r)$.

Proof. Induction over the length of the list ps. If the length is $\perp$ then we use a separate lemma to show that `splitM` must also return $\perp$. If $ps = []$ then that p will be either ALLRIGHT or ALLLEFT depending on the value of $f\ s$. In the inductive case, $ps = ((s',p'):ps')$. If $s = s'$ then let $p = p'$. Otherwise use IH to show that some p exists, because the modifications to the context mapped to by s' will not affect those of s. □

Theorem 5.1.2. *If* $\mathsf{PRE}_{\mathtt{s}}$ *then* $\mathsf{PRE}_{\texttt{smap}\ \mathtt{s}}$.

Proof. We use Lemma 5.1.2 to show that if `splitM` splits a map c of type SMap (Cxt ς) into two separate maps c_l and c_r then the context associated with each individual string will have been split with `pfCxt`.

$c_l \sqsubseteq_{\texttt{smap}\ \mathfrak{s}} c$ and $c_r \sqsubseteq_{\texttt{smap}\ \mathfrak{s}} c$ can then be shown directly. If an action is permitted by the sub-context then either the context for that particular string is identical to the parent context, in the case of AllLeft/AllRight, or it was split using Split p, for some p. In this case $\mathsf{PRE}_{\mathfrak{s}}$ may be applied.

To prove that $c_l \Diamond_{\texttt{smap}\ \mathfrak{s}} c_r$, we do case analysis on whether the left-hand and right-hand actions (s_1, a_1) and (s_2, a_2) affect different worlds. In other words, is $s_1 = s_2$ or not? If $s_1 = s_2$ then it is impossible that s_1's context was split with either AllLeft or AllRight since that would render one or the other action disallowed. We must use $\mathsf{PRE}_{\mathfrak{s}}$ to show that the actions are order independent. If $s_1 \neq s_2$ then this will be true automatically because both actions will be affecting different parts of the world state, and therefore won't interfere with each other. □

As an example, imagine a basic I/O model called $\mathfrak{file}$ which describes the contents of a file and the basic actions one could perform on it, such as `fOpen`, `fClose`, etc. The I/O model $\texttt{smap}\ \mathfrak{file}$ can then be used as an effective model of a file system, (and this is exactly what is done in Chapter 6).

The program `wordCounter` ns counts the total number of words within each file in the list ns of filenames by concurrently running a separate `wordCount` program on each individual file. If that file is locked out by the current context then `wordCount` returns 0. This may happen if a particular filename occurs twice in ns, or if `wordCounter` is executed in parallel with other processes which have already claimed access to a particular file that is wanted.

```
wordCounter :: [String] → Prog_{smap file} Int
wordCounter []     = return 0
wordCounter (n:ns) = par [(n,AllLeft)] (wordCount n) (wordCounter ns) (+)


wordCount :: String → Prog_{smap file} Int
wordCount fname = test (fname,fOpen)
  (do
    action (fname,fOpen)
    -- ...
    action (fname,fClose)
    return i)
  (return 0)
```

5.1.3 Other possible combinators

The reader will probably have noticed that some combinators, particularly $*$, tend to make the API somewhat ugly. Our actions and return values become things like Left Rcve and Right 3 and it no longer looks like a real API. The appearance of an API is important if people are to understand how it corresponds to real system calls.

This could be solved by adding 'cosmetic' combinators which perform alpha conversion on the various types, turning them into something more realistic. For example,

$$\texttt{alphaa} \;::\; (\alpha \to \alpha_1) \to \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \to \mathsf{IOModel}\ \nu\ \alpha_1\ \rho\ \omega\ \varsigma$$
$$\texttt{alphav} \;::\; (\nu \to \nu_1) \to \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \to \mathsf{IOModel}\ \nu_1\ \alpha\ \rho\ \omega\ \varsigma$$

We could do this but we decide not to. We instead just write library functions which wrap the actual API calls in a pretty, usable interface and this works just as well.

Sometimes the function `pf`, which is used to split contexts in an I/O model, can behave in a way which is not symmetric. If `pf` $p\ c = (c_l, c_r)$ then there may not necessarily be a p' such that `pf` $p'\ c = (c_r, c_l)$. This shortcoming becomes serious when a smaller I/O model is modified using combinators. Imagine an I/O model $\mathfrak{lbfr}$ which always forces the left-hand process to receive and the right-hand process to send. While $\mathfrak{lbfr}$ is only slightly less powerful than $\mathfrak{bfr}$ on its own, the model $\mathfrak{lbfr} * \mathfrak{lbfr}$ is severely limited compared to $\mathfrak{bfr} * \mathfrak{bfr}$: the left-hand process will only be allowed to send to both buffers and the right hand will only be allowed receive from both buffers.

It is not difficult to develop a combinator that gets around this by adding an extra Boolean bit to the ρ type. This indicates if the left and right contexts should be swapped.

5.2 Attempting dynamic allocation

5.2.1 The problem

The string map combinator allows one to safely control concurrent access to multiple resources. Its downside is that we first need to know the string which identifies that resource before any concurrency takes place. This is not a problem when controlling access to entire files, but consider the act of opening a file for reading, or allocating memory on the heap. In these two cases an API call allocates a new structure (an internal read handle; memory) and returns information about how this structure may be accessed (a file descriptor; a memory address).

Ideally we would like to develop a combinator which was similar to the string map combinator with the exception that it also defined a new, special API call. This new action would create a structure on the fly, a buffer, say, and return information to indicate how this buffer can be accessed.

This is what we would ideally like but there are some immediate difficulties.

1. Two concurrent processes should be able to allocate new buffers without affecting one another. If they do, then how can we ensure that the returned handles and the world state will be identical even if the interleaving of threads differs?

2. When performing concurrency, we must ensure that each process is able to access the

resources it may want to use. In this case resources are identified by handles, so how do we know in advance which handles each process may need access to?

3. Assuming a process has no control over the allocated handle, what happens if a process allocates a handle which it does not have access to? I/O contexts can only prevent actions taking place based on their *arguments* not their possible return values.

Number (1) could be said to relate to our language semantics which layers an outer operational semantics on top of an inner denotational semantics for the host language. Although handles may be internally represented by, perhaps, an integer, that actual integer value is irrelevant. As long as handles are used in a normal fashion then there should be no difficulties. Similarly, if the internal world state is a list of buffers, then the ordering shouldn't matter if we don't have any means of distinguishing it as a programmer.

Unfortunately this sort of fine-grained control over what the programmer can and cannot do, or should and should not do, is beyond the limits of the semantics. Denotationally, a function of the type $\mathsf{Handle} \to \mathsf{Prog}_{\mathfrak{s}}$ can be any mathematical function from a handle (whatever it may be) to a resultant program. Once we accept that certain handles are different to others, we must also accept that, without completely altering the denotational and operational semantics of the host language, a program can then distinguish them whatever way it likes.

5.2.2 A basic solution

As an initial solution, we at first describe a weak dynamic combinator.

We leave problem (1) unsolved by forcing allocation to be explicitly sequenced. However once a structure has been allocated then any amount of concurrency may take place on it. The second problem becomes something that the programmer must look after him/herself. One can still easily give a process the right to allocate structures without giving it access to any of those structures. Our solution to problem (3) is to turn allocation into two separate steps, as we shall now show.

The new `dmap` combinator has the following type:

$$\begin{aligned}\texttt{dmap} \;::\; & \omega \to \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \to \\ & \mathsf{IOModel}\ (\mathsf{Either}\ \nu\ \mathsf{Int})\ (\mathsf{DynAction}\ \mathsf{Int}\ \alpha)\ [(\mathsf{Int}, \mathsf{Splitter}\ \rho)]\ [\omega]\ (\mathsf{Bool}, \mathsf{Int} \to \mathsf{Cxt}\ \varsigma)\end{aligned}$$

Handles are represented by Ints. World state becomes a list of ω and the allocation of new structures must be single threaded. Each action is an element of the $\mathsf{DynAction}\ \mathsf{Int}\ \alpha$ type.

$$\mathsf{DynAction}\ \iota\ \alpha \;\triangleq\; \textsc{DynAct}\ \iota\ \alpha \mid \textsc{Next} \mid \textsc{Alloc}\ \iota$$

One can either do

- DynAct i a, which performs action a on the structure identified by i.
- Next, which retrieves the next free index in the list.
- Alloc i, which allocates a new structure at index i, failing if it is not the next free index. The first argument to the `dmap` combinator gives the initial value of this new structure.

The return value type becomes $\mathsf{Either}\ \nu\ \mathsf{Int}$ because an action DynAct i a must return a ν, and Next must return an Int. (Alloc i returns some fixed Int which the programmer will presumably just ignore).

The context is a tuple of type $(\mathsf{Bool}, \mathsf{Int} \to \mathsf{Cxt}\ \varsigma)$. If the first Bool is False then a program running in that context cannot perform either Next or Alloc. If a context permits allocation, then only the right-hand sub-context can inherit this capability. This guarantees single threaded allocation by ensuring that at most one concurrent sub-program can perform Next and Alloc.

The map of type $\mathsf{Int} \to \mathsf{Cxt}\ \varsigma$ indicates the permissions associated with the structure at each individual index, and it is this which determines whether an action DynAct i a is allowed. These maps are split using `splitM`. Crucially, however, one can only perform Alloc i if, as well as being the only sub-process that is allowed to allocate, one has *complete* access to index i. In other words, the context associated with index i must be the context which permits all actions. It is only by doing this that we can guarantee that another process cannot accidentally interfere with the allocation of that index. A parallel process with partial access to that index could be affected by the creation of a new structure.

The world state is manipulated with these operations:

$$\begin{aligned}
(\texttt{!!}) &:: \forall\beta.\ [\beta] \to \mathsf{Int} \to \beta \\
\texttt{ovwL} &:: \forall\beta.\ \mathsf{Int} \to \beta \to [\beta] \to [\beta] \\
\texttt{newL} &:: \forall\beta.\ \mathsf{Int} \to \beta \to [\beta] \to [\beta] \\
\texttt{stL} &:: \forall\beta.\ \forall\delta.\ (\beta \to (\beta, \delta)) \to \mathsf{Int} \to [\beta] \to ([\beta], \delta)
\end{aligned}$$

`(!!)` is the standard Haskell list look-up operation. `ovwL` i b bs overwrites the i'th element of the list bs with b, failing if i is out of bounds. `newL` i b bs concatenates element b to the end of the list bs succeeding only if i equals the length of bs. The function `stL` is similar to `stM`. `stL` f i is a state-transformer which modifies the i'th element of a list according to f, returning f's return value.

Lemma 5.2.1.

1. *If* $i_1 \neq i_2$ *then* $(\texttt{ovwL}\ i_2\ v\ l)\ \texttt{!!}\ i_1 = l\ \texttt{!!}\ i_1$.
2. $(\texttt{ovwL}\ i\ v\ l)\ \texttt{!!}\ i = v$

```
applySnd :: (γ → δ) → (β,γ) → (β,δ)        isTopCxt :: Cxt ς → Bool
applySnd f (a,b) ≙ (a, f b)                isTopCxt CxtB ≙ False
                                           isTopCxt (Cxt c b) ≙ b

  dmap w0 ⟨af, wa, ap, pf⟩ ≙ ⟨
      λla. λwl. case la of
          DynAct i a →   applySnd Left (stL (af a) i wl)
          Next →         (wl, Right (length wl))
          Alloc i →      (newL i w0 wl, Right 0),
      λla. λwl. case la of
          DynAct i a →   wa a (wl !! i)
          Next →         False
          Alloc i →      False,
      λ(b,cm). λla. case la of
          DynAct i a →   apCxt ap (lkpM i cm) a
          Next →         b
          Alloc i →      b && isTopCxt (lkpM i cm),
      λps. λ(b,cm). case (splitM pf (λc. False) ps cm) of
          (cml,cmr) →    ((False,cml),(b,cmr))
  ⟩
```

Figure 5.3: Definition of `dmap`

3. $\mathit{ovwL}\ i\ v_1\ (\mathit{ovwL}\ i\ v_2\ l) = \mathit{ovwL}\ i\ v_1\ l$

4. $\mathit{length}\ (\mathit{ovwL}\ i_1\ v\ l) = \mathit{length}\ l$.

5. *If* $i_1 \neq i_2$ *then* $\mathit{ovwL}\ i_1\ v_1\ (\mathit{ovwL}\ i_2\ v_2\ l) = \mathit{ovwL}\ i_2\ v_2\ (\mathit{ovwL}\ i_1\ v_1\ l)$

6. $\mathit{ovwL}\ i_1\ v_1\ (\mathit{newL}\ i_2\ v_2\ l) = \mathit{newL}\ i_2\ v_2\ (\mathit{ovwL}\ i_1\ v_1\ l)$

7. *If* $i_1 \neq i_2$ *then* $\mathsf{cmmt}(\mathit{stL}\ f_1\ i_1, \mathit{stL}\ f_2\ i_2, l)$.

8. *If* $\mathsf{cmmt}(f_1, f_2, l\ \mathit{!!}\ i)$ *then* $\mathsf{cmmt}(\mathit{stL}\ f_1\ i, \mathit{stL}\ f_2\ i, l)$

9. $\mathsf{cmmt}(\mathit{stL}\ f_1\ i_1, (\lambda w.\ (\mathit{newL}\ i_2\ v_2\ w, x)), l)$.

Proof. The first six are proved by straightforward induction, and the last three use these results to show how state-transformers on individual list elements may commute. The interaction between `newL` and other operations is probably the most subtle aspect to these proofs. `newL` is unaffected by `ovwL` and `stL` only because `newL` appends one element to the end of the list, and `ovwL`/`stL` only modify existing elements of the list. □

The full definition of `dmap` is given in Figure 5.3 and we now show that it preserves confluence.

Theorem 5.2.1. *If* $\mathsf{PRE}_{\mathfrak{s}}$ *then* $\mathsf{PRE}_{\mathit{dmap}\ w_0\ \mathfrak{s}}$.

Proof. Like with `smap` we must apply Lemma 5.1.2 to show that the left- and right-hand sub-contexts have been split correctly for each index. We also have the added knowledge that at most one sub-context can allow allocation.

$c_l \sqsubseteq_{\texttt{dmap}\ \mathfrak{s}} c$ and $c_r \sqsubseteq_{\texttt{dmap}\ \mathfrak{s}} c$ can be shown in a similar way to that of `smap` with the added knowledge that if a sub-context permits allocation then the parent context must also.

Proving $c_l \lozenge_{\texttt{dmap}\ \mathfrak{s}} c_r$ is somewhat more complex since we have three different types of action. Actions relating to allocation cannot run concurrently with one another so there are only three possible pairs which we have to prove for:

- DynAct i_l a_l and DynAct i_r a_r. If $i_l \neq i_r$ then the actions are order independent regardless of context information because each action affects a different index – Lemma5.2.1 (1) and (7). If $i_l = i_r$ then we must apply $\mathsf{PRE}_{\mathfrak{s}}$ using the knowledge that the context associated with $i_l (= i_r)$ was split in a way which makes a_l and a_r commute – Lemma5.2.1 (2) and (8).

- Next and DynAct i_r a_r. Next $|||_{\texttt{dmap}\ w_0\ \mathfrak{s}}$ DynAct i_r a_r holds because Next does not change world state, and action DynAct i_r a_r cannot change the length of the list. $\mathsf{ally}_{\texttt{dmap}\ w_0\ \mathfrak{s}}$(Next, DynAct i_r a_r) is true because Next doesn't change world state, and $\mathsf{ally}_{\texttt{dmap}\ w_0\ \mathfrak{s}}$(DynAct i_r a_r, Next) is true trivially because Next is never stalled.

- Alloc i_l and DynAct i_r a_r. The proof of Alloc i_l $|||_{\texttt{dmap}\ w_0\ \mathfrak{s}}$ DynAct i_r a_r relates to the fact that i_l cannot equal i_r, because for Alloc i_l to be permitted it must have complete access to index i_l, thus forbidding any other actions. Once $i_l \neq i_r$ is known we use the fact that `newL` doesn't affect `stL`. Similarly, $\mathsf{ally}_{\texttt{dmap}\ w_0\ \mathfrak{s}}$(Next, DynAct i_r a_r) holds because `newL` will not affect index i_r, thus not causing DynAct i_r a_r to become stalled. $\mathsf{ally}_{\texttt{dmap}\ w_0\ \mathfrak{s}}$(DynAct i_r a_r, Alloc i_l) is true trivially because Alloc i_l is never stalled.

□

5.2.3 Usage of `dmap`

Figure 5.4 defines three helper functions which are of assistance when writing programs for I/O models constructed directly using the `dmap` combinator. These are introduced for the reader's benefit – they will not be used outside of this chapter.

- `allocateD` attempts to allocate a new structure, whatever it may be, returning a new index to that structure. It performs both the Next and the Alloc actions, and returns Nothing if one or both are not permitted by the program's context.

- `actionD` i a performs action a on the structure at index i, where it is assumed that α and ν are the types of actions and return values respectively in I/O model $\mathfrak{s}$.

```
allocateD :: Prog_{dmap w0 s} (Maybe Int)
allocateD = test Next
  (do
     Right i <- action Next
     test (Alloc i)
       (do
          action (Alloc i)
          return (Just i))
       (return Nothing))
  (return Nothing)
```

```
actionD :: Int → α → Prog_{dmap w0 s} ν
actionD i a = do
  Left v <- action (DynAct i a)
  return v

testD :: Int → α → Prog_{dmap w0 s} Bool
testD i a = test (DynAct i a)
  (return True)
  (return False)
```

Figure 5.4: Helper functions for the `dmap` combinator

- `testD` i a returns a Bool which indicates if `actionD` i a is permitted by the current context.

The following example program creates two buffers, spawns a new process which performs a separate task and then proceeds, communicating with that new process. The new process receives an integer i from one buffer and returns $i * 2$ on the other buffer. This continues until it receives -1.

```
bffrProgram :: Prog_{dmap [] bffr} ()
bffrProgram = do
  Just i1 <- allocateD                  -- allocate two new buffers
  Just i2 <- allocateD                  --
  par                                   -- LHS: can send on i1 & rcve on i2
    [(i1,Split True),(i2,Split False)] -- RHS: can do everything else
    (doubleBffr i2 i1)                  -- run doubleBffr using i1 i2
    (restOfProgram i1 i2)
    (\_ _ -> ())


doubleBffr :: Int → Int → Prog_{dmap [] bffr} ()
doubleBffr inB outB = do
  i <- actionD inB Rcve          -- receive an integer 'i' from 'in'
  if (i==-1)                     -- if i=-1 finish, otherwise
    then (return ())             -- send i*2 and repeat
    else (actionD outB (Send (i*2)) >> doubleBffr inB outB)
```

Of course, this is a terrible use of buffers in its own right. It could have been done in a purely functional style without requiring monads at all. But if `doubleBffr` also needed access to a particular file then this would have been out of the question, and we will show more practical uses such as this in Chapter 6.

We can use this example to discuss a few subtleties regarding the `dmap` combinator.

- Since the two buffers have just been allocated we can be certain that they are empty. We usually do not know the initial state of part of an I/O model because a program, in general, cannot control the world state with which it interacts initially.

- If `doubleBffr` needed to create internal communication buffers then this would not be possible. With `dmap` allocation must be single threaded, and we have chosen to give `restOfProgram` this right.

- The buffers identified by `i1` and `i2` are neither local nor lambda-bound. Once created they will exist as a part of global state for the duration of the program and may be reused if the sub-program's context permits it.

  We don't consider deallocation, garbage collection or the re-use of indices to be semantically interesting. If one needs to model deallocation and the fact that actions on a deallocated buffer will fail, then this should be incorporated as part of the 𝔟𝔣𝔣𝔯 model itself. In this situation the world state would become a buffer and a value to indicate if the buffer is no longer to be used (see, for example, 𝔠𝔥𝔞𝔫 in Section 6.3.1).

5.3 Location-based I/O models

The `dmap` combinator defined above is not entirely satisfactory. Its chief fault is that allocation must be sequenced explicitly. For small examples this may suffice but it does not scale well. Moreover, it is at odds with our intuition – surely the allocation of arrays or read handles isn't really affected by concurrency just as long as access to each new resource is properly controlled once it is created?

Single threaded allocation is necessary because as long as we have one single API call for allocation then it cannot distinguish the needs of two different concurrent processes. The returned handles must always be distinguishable – after all, they refer to distinct structures – and the new structures must occupy a unique places in a global state. This means that two processes calling that action will always affect one another.

In this section we present the seeds of a solution to the above difficulty. We do this by re-thinking the sentence "one single API call ... cannot distinguish the needs of two different concurrent processes". What if each process could calculate, at runtime, a piece of knowledge that allowed it to distinguish itself from every other process currently running? By passing this as an argument to the allocation action, and using a suitable data-structure, one could then allow any process to allocate when it wants.

5.3.1 The approach

Our solution begins by finding a way of identifying processes at runtime. Since a program is a tree of concurrent processes, each process can be identified by the route to that leaf.

The structure Loc, which we refer to as "locations", does this:

$$\begin{aligned} \mathsf{Dir} &\triangleq \mathrm{L} \mid \mathrm{R} \\ \mathsf{Loc} &\triangleq [\mathsf{Dir}] \end{aligned}$$

$l_1 \preccurlyeq l_2$ indicates that location l_1 is a prefix of the location l_2. For example, $[\mathrm{L}, \mathrm{R}] \preccurlyeq [\mathrm{L}, \mathrm{R}, \mathrm{L}]$. The relation $\preccurlyeq$ is a partial order (reflexive, anti-symmetric and transitive) and has the following characteristic property:

$$l_1 \preccurlyeq l_2 \quad \Leftrightarrow \quad \exists_{l_3 \in \mathsf{Loc}}. l_1 +\!\!+ l_3 = l_2$$

We can now propose the following possible approach.

- The context becomes a tuple of the form $(\mathsf{Loc}, \varsigma)$.
- When we split a context (l, c) this becomes contexts $(l +\!\!+ [\mathrm{L}], c_l)$ and $(l +\!\!+ [\mathrm{R}], c_r)$.
- An action which needs location information must have its location supplied explicitly as a parameter. This is to ensure that the action can use this location when changing world state and also to guarantee that if one supplies an illegal location then the action will be disallowed.
- A process must be able to determine the location in which it is executing. The `test` command can extract a boolean value from the current context (i.e. whether a particular action is permitted), so we must use this multiple times to calculate the entire location.

This is roughly the solution that we adopt but there are two notable problems with this:

- The I/O context only needs to record location information once. In the presence of combinators this may end up being needlessly duplicated. While there is nothing technically wrong with the location being duplicated, it is confusing (what if the two locations are different?) and unnecessary.
- The new API call will no longer look like a real system call. The appearance of an API is not of great importance, as mentioned earlier, since it can be tidied up with libraries. But this is going further because to make it look realistic a whole argument to the API call would then have to be completely invisible to the user.

To solve this we propose a new I/O model $\mathsf{IOModelL}\ \nu\ \alpha\ \rho\ \omega\ \varsigma$ which is closely related to $\mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega\ \varsigma$ except that locations are in-built as part of its definition. In this I/O model certain actions are location-sensitive and, as well as the normal constraints imposed by contexts, two location-sensitive actions cannot be executed in the same location. We give a new precondition $\mathsf{PRE}'_{\mathfrak{s}}$ for this type of location-based I/O model, give conversion

$$
\begin{array}{rcl}
\mathfrak{s} :: \mathsf{IOModelL}\ \nu\ \alpha\ \rho\ \omega\ \varsigma & \triangleq & \langle \mathtt{af}' :: (\mathsf{Loc}, \alpha) \to \omega \to (\omega, \nu), \\
& & \ \mathtt{wa}' :: (\mathsf{Loc}, \alpha) \to \omega \to \mathsf{Bool}, \\
& & \ \mathtt{ap}' :: \varsigma \to \alpha \to \mathsf{Bool}, \\
& & \ \mathtt{pf}' :: \mathsf{Loc} \to \rho \to \varsigma \to (\varsigma, \varsigma), \\
& & \ \mathtt{ig}' :: \alpha \to \mathsf{Bool} \rangle \\
\\
\lozenge'_{\mathfrak{s}} & : & \varsigma \to \varsigma \to \mathbb{B} \\
c_l \lozenge'_{\mathfrak{s}} c_r & \triangleq & \forall_{a_l \in \alpha}.\forall_{l_l \in \mathsf{Loc}}.\forall_{a_r \in \alpha}.\forall_{l_r \in \mathsf{Loc}}. \\
& & \quad \mathtt{ap}'\ c_l\ a_l \wedge \mathtt{ap}'\ c_r\ a_r \wedge (\mathtt{ig}'\ a_l = \textsc{True} \vee \mathtt{ig}'\ a_r = \textsc{True} \vee l_l \neq l_r) \Longrightarrow \\
& & \quad\quad (l_l, a_l) \mid\mid\mid_{\mathfrak{s}} (l_r, a_r) \wedge \mathsf{ally}_{\mathfrak{s}}((l_l, a_l), (l_r, a_r)) \wedge \mathsf{ally}_{\mathfrak{s}}((l_r, a_r), (l_l, a_l)) \\
\mathsf{PRE}'_{\mathfrak{s}} & \triangleq & \forall_{l \in \mathsf{Loc}}.\forall_{p \in \rho}.\forall_{c \in \varsigma}.\forall_{c_l \in \varsigma}.\forall_{c_r \in \varsigma}. \\
& & \quad \mathtt{pf}'\ l\ p\ c = (c_l, c_r) \Longrightarrow c_l \sqsubseteq_{\mathfrak{s}} c \wedge c_r \sqsubseteq_{\mathfrak{s}} c \wedge c_l \lozenge'_{\mathfrak{s}} c_r
\end{array}
$$

Figure 5.5: Definition of location-based I/O models

functions which convert one I/O model type to the other, and show that these preserve the respective pre-conditions.

The central idea is that locations need to be added to the context only once. Any normal I/O model may be made location-based by denoting every action as one that does not care about its location. We can then combine many location-based models together using various combinators, constructing a useful large I/O model, and only at the end do we convert it back to a traditional model.

5.3.2 Definition and relationship to normal I/O models

Location-based I/O models and their respective pre-condition are defined in Figure 5.5.

This structure differs in that

- In the API, the effect of each action on world state is determined also by another argument of type Loc. Hence the types of the functions `af'` and `wa'` differ from that of `af` and `wa`.
- There is a new function `ig'` (short for 'ignore') which indicates if an action should ignore the extra location. If `ig'` a = False then action a is said to be "location-sensitive".
- The function `pf'` which splits contexts also takes an additional Loc argument. This is handy because sometimes the current location will affect the way in which contexts are split, as will become clearer later in the chapter.

$\mathsf{PRE}'_{\mathfrak{s}}$ is mostly similar to $\mathsf{PRE}_{\mathfrak{s}}$. The definitions of $\mathsf{ally}_{\mathfrak{s}}$, $\sqsubseteq$ and $\mid\mid\mid_{\mathfrak{s}}$ remain unchanged, and the only extra condition is that given two actions (l_l, a_l) and (l_r, a_r), if a_l and a_r

```
DAction α  ≙  PROBE | ACT α
   MtoLM   ::  IOModel ν α ρ ω ς → IOModelL ν α ρ ω ς
   LMtoM   ::  IOModelL ν α ρ ω ς → IOModel ν (Loc,DAction α) ρ ω (Loc,ς)

                                  LMtoM ⟨af′, wa′, ap′, pf′, ig′⟩ ≙ ⟨
MtoLM ⟨af, wa, ap, pf⟩ ≙ ⟨            λ(l,ACT a). λw. af′ (l,a) w,
    λ(l,a). λw. af a w,                λ(l,ACT a). λw. wa′ (l,a) w,
    λ(l,a). λw. wa a w,                λ(l_c,c). λ(l_a,a). case a of
    ap,                                    ACT a →        ap′ c a && (ig′ a || l_c ≼ l_a)
    λl. pf,                                PROBE →        l_c ≼ l_a,
    λa. TRUE                           λp. λ(l,c). case (pf′ l p c) of
⟩                                          (c_l,c_r) →    ((l⧺[L],c_l),(l⧺[R],c_r))
                                   ⟩
```

Figure 5.6: Converting between IOModel and IOModelL

are both location-sensitive then it is assumed that $l_l \neq l_r$ hold before we try to prove the non-interference property.

The two functions for converting between the two I/O models are in Figure 5.6.

The function MtoML converts a traditional I/O model to a location-based one. The conversion is relatively simple – no action becomes location-sensitive, and all location information is ignored.

Theorem 5.3.1. *If* $\mathsf{PRE}_{\mathfrak{s}}$ *then* $\mathsf{PRE}'_{\mathit{MtoLM}\,\mathfrak{s}}$.

Proof. If $\mathtt{pf}'\ l\ p\ c = (c_l, c_r)$ then $\mathtt{pf}\ p\ c = (c_l, c_r)$, so we know from $\mathsf{PRE}_{\mathfrak{s}}$ that $c_l \sqsubseteq_{\mathfrak{s}} c$, $c_r \sqsubseteq_{\mathfrak{s}} c$ and $c_l \lozenge_{\mathfrak{s}} c_r$. $c_l \sqsubseteq_{\mathtt{MtoLM}\,\mathfrak{s}} c$ and $c_r \sqsubseteq_{\mathtt{MtoLM}\,\mathfrak{s}} c$ are immediately true. $c_l \lozenge'_{\mathtt{MtoLM}\,\mathfrak{s}} c_r$ follows because, since no action is location-sensitive, the extra condition $\mathtt{ig}'\ a_l \vee \mathtt{ig}'\ a_r \vee l_l \neq l_r$ in PRE' always holds, and the new state-transformer behaves identically because it ignores l_l and l_r. □

The conversion of an IOModelL back to an IOModel using LMtoM is a little more difficult to explain and motivate. When we convert an IOModel into an IOModelL location information is introduced. When converting back, however, this information is not removed, it is merely made explicit in the model. It is best to think of LMtoM as giving the semantics of location-based I/O models (a new concept) in terms of normal I/O models (an established concept). In these converted models certain aspects of actions and I/O contexts are hardwired to behave in a special way.

The conversion requires the insertion of a Loc into the context, and the requirement that all actions have an extra Loc as a parameter. As for the actions themselves, another action PROBE is included when converting back to a IOModel. This should never be executed (it always fails) but the action (*l*,PROBE) is *permitted* if $l_c \preccurlyeq l$, where l_c is the current location.

Using `test` this lets a process query its location. Other actions, those of the form $(l, \text{ACT } a)$, are permitted by context (l_c, c) only if $\texttt{ap}'\ c\ a$, and if action a is location-sensitive then $l_c \preccurlyeq l$ must also hold.

When context (l, c) is split with p then the current location l is given to the function $\texttt{pf}'$ as well as p and c. If $\texttt{pf}'\ l\ p\ c = (c_l, c_r)$ then the resultant left- and right-hand contexts become $(l{+\!\!+}[\text{L}], c_l)$ and $(l{+\!\!+}[\text{R}], c_r)$.

The following important result shows that this conversion preserves confluence:

Theorem 5.3.2. *If* $\mathsf{PRE}'_{\mathfrak{s}}$ *then* $\mathsf{PRE}_{\mathit{LMtoM}\ \mathfrak{s}}$.

Proof. If the context (l, c) has successfully been split into $(l{+\!\!+}[L], c_l)$ and $(l{+\!\!+}[R], c_r)$ then $\texttt{pf}'\ l\ p\ c = (c_l, c_r)$. This is applied to $\mathsf{PRE}'_{\mathfrak{s}}$ to prove $c_l \sqsubseteq_{\mathfrak{s}} c$, $c_r \sqsubseteq_{\mathfrak{s}} c$ and $c_l \lozenge'_{\mathfrak{s}} c_r$. We use this to prove the following:

- $(l{+\!\!+}[L], c_l) \sqsubseteq_{\texttt{LMtoM}\ \mathfrak{s}} (l, c)$. If action $(l_l, \text{ACT } a_l)$ is permitted by context $(l{+\!\!+}[L], c_l)$ then a_l must be permitted by context c_l, and is either not location-sensitive or $l{+\!\!+}[L] \preccurlyeq l_l$. Therefore, by $c_l \sqsubseteq_{\mathfrak{s}} c$, a_l is also permitted by context c_l and if action a_l is location-sensitive then $l \preccurlyeq l_l$ can be proved from $l{+\!\!+}[L] \preccurlyeq l_l$.

  If (l_l, PROBE) is permitted by context $(l{+\!\!+}[L], c_l)$ then $l{+\!\!+}[L] \preccurlyeq l_l$, which implies $l \preccurlyeq l_l$, proving that (l_l, PROBE) is permitted by context (l, c_l).

- $(l{+\!\!+}[R], c_r) \sqsubseteq_{\texttt{LMtoM}\ \mathfrak{s}} (l, c)$. Similar to the proof for c_l.

- $(l{+\!\!+}[L], c_l) \lozenge_{\texttt{LMtoM}\ \mathfrak{s}} (l{+\!\!+}[R], c_R)$. If either the left-hand or right-hand action is of the form (l_l, PROBE) then this holds trivially, because PROBE never successfully executes. To prove this for any actions $(l_l, \text{ACT } a_l)$ and $(l_r, \text{ACT } a_r)$ it must be shown that $\texttt{ig}'\ a_l \vee l{+\!\!+}[L] \preccurlyeq l_l$ and $\texttt{ig}'\ a_r \vee l{+\!\!+}[R] \preccurlyeq l_r$ (from the definition of `ap` in `LMtoM`) together imply $\texttt{ig}'\ a_l \vee \texttt{ig}'\ a_r \vee l_l \neq l_r$ (from the definition of PRE'). This becomes a proof that $l{+\!\!+}[L] \preccurlyeq l_l$ and $l{+\!\!+}[R] \preccurlyeq l_r$ implies $l_l \neq l_r$, which requires one or two small lemmas. The actions can then be shown to be order independent using $c_l \lozenge'_{\mathfrak{s}} c_r$.

□

The action PROBE can be used by a process to determine its location at runtime with the small program $\texttt{probeLoc} :: \mathsf{Prog}_{\texttt{LMtoM}\ \mathfrak{s}}\ \mathsf{Loc}$ whose implementation is given in Section A.2. It is unfortunately extremely inefficient, but it does what is required of it.

Assume a location-based I/O model $\mathfrak{s} :: \mathsf{IOModelL}\ \nu\ \alpha\ \rho\ \omega\ \varsigma$. After converting $\mathfrak{s}$ back to a standard model, we can now both perform an action and check if an action is permitted by the current context in a "normal" way (i.e. making location information transparent):

```
actionL :: α → Prog_{LMtoM s} ν
actionL a = probeLoc >>= \l -> action (l,Act a)
```

```
testL :: α → Prog_(LMtoM s) Bool
testL a = probeLoc >>= \l -> test (l,Act a) (return True) (return False)
```

Since location-based I/O models have been shown to be inter-convertible with the existing models it perhaps is a suitable candidate as a full replacement. There is a messy loop-back in this approach: a process must determine its location, give it as an argument to a particular action and then (assuming that we want the use of locations to be transparent) discard it. We will not consider it further in this dissertation, but perhaps this should be automated entirely using only location-based I/O models.

5.3.3 Simple location-based combinators

The combinators $*$ and `smap` also have location-based equivalents, $*'$ and `smap'`. The types of these combinators are extremely similar and their omitted definitions are also largely equivalent, requiring few changes. The extra `Loc` argument to `af'`, `wa'` and `pf'` is passed unchanged to the respective functions of the I/O models being modified, and an action remains location-sensitive if it instead modifies a small part of a much larger world state.

$$
\begin{aligned}
(*') \quad &:: \quad \mathsf{IOModelL}\ \nu_1\ \alpha_1\ \rho_1\ \omega_1\ \varsigma_1 \rightarrow \mathsf{IOModelL}\ \nu_2\ \alpha_2\ \rho_2\ \omega_2\ \varsigma_2 \\
&\qquad \rightarrow \mathsf{IOModelL}\ (\mathsf{Either}\ \nu_1\ \nu_2)\ (\mathsf{Either}\ \alpha_1\ \alpha_2)\ (\rho_1, \rho_2)\ (\omega_1, \omega_2)\ (\varsigma_1, \varsigma_2) \\
\mathtt{smap'} \quad &:: \quad \mathsf{IOModelL}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \rightarrow \\
&\qquad \mathsf{IOModelL}\ \nu\ (\mathsf{String}, \alpha)\ [(\mathsf{String}, \mathsf{Splitter}\ \rho)]\ (\mathsf{SMap}\ \omega)\ (\mathsf{SMap}\ (\mathsf{Cxt}\ \varsigma))
\end{aligned}
$$

These all preserve confluence as before. We omit the proofs but like all other results in this chapter they have been machine-verified.

Theorem 5.3.3. *If* $\mathsf{PRE}'_{\mathfrak{s}_1}$ *and* $\mathsf{PRE}'_{\mathfrak{s}_2}$ *then* $\mathsf{PRE}'_{\mathfrak{s}_1 *' \mathfrak{s}_2}$.

Theorem 5.3.4. *If* $\mathsf{PRE}'_{\mathfrak{s}}$ *then* $\mathsf{PRE}'_{smap'\ \mathfrak{s}}$.

Also, the following three lemmas show that these combinators are related to the old combinators in a natural way.

Lemma 5.3.1. $MtoLM\,(\mathfrak{s}_1 * \mathfrak{s}_2) = (MtoLM\,\mathfrak{s}_1) *' (MtoLM\,\mathfrak{s}_2)$

Lemma 5.3.2. $MtoLM\,(smap\,\mathfrak{s}) = smap'\ (MtoLM\,\mathfrak{s})$

One might ask if we could prove that, for example, `LMtoM (smap' s) = smap (LMtoM s)`. This is not true and it demonstrates one of the very reasons we introduced location-based I/O models. The I/O model `LMtoM (smap' s)` contains location information just once but with `smap (LMtoM s)` the context contains an infinite number of locations, one for each `String`.

Also, `MtoLM (LMtoM s) ≠ s`. These two models are certainly equivalent in some sense but the context in the model on the left contains extra redundant location information, none of which has any effect on the behaviour of actions.

5.4 The dmap′ combinator

Having defined location-based I/O models, the dmap combinator can now be re-written so that it lets many processes allocate concurrently. We use the extra Loc argument in conjunction with a special data-structure to guarantee the non-interference of two allocation commands.

The new combinator dmap′ has the following type:

$$\begin{aligned}\texttt{dmap}' \;::\; & \omega \to \mathsf{IOModelL}\ \nu\ \alpha\ \rho\ \omega\ \varsigma \to \\ & \mathsf{IOModelL}\ (\mathsf{Either}\ \nu\ \mathsf{HndP})\ (\mathsf{DynAction}\ \mathsf{HndP}\ \alpha) \\ & [(\mathsf{HndP}, \mathsf{Splitter}\ \rho)]\ (\mathsf{Pool}\ \omega)\ (\mathsf{HndP} \to \mathsf{Cxt}\ \varsigma)\end{aligned}$$

5.4.1 Pools

The Pool β structure, used as the world state in dmap′, is in effect a binary tree of lists and it replaces the plain lists used in dmap to store the newly allocated structures. With dmap there was just one list and any new structure was appended to this list. The new combinator dmap′ creates an I/O model in which world state is partitioned into many different lists in such a way that each process, when allocating, can modify its own unique list.

Once again, like with dmap, the newly created structure will still be global and may be modified and accessed long after the process which created it has gone. This partitioning exists only to ensure that if two processes are concurrently allocating two new structures then the interleaving of threads will in no way affect the resultant handles or the final world state.

A Loc identifies the route to a node in the Pool tree, and an Int identifies a particular element of that list. Therefore individual elements of Pool β are identified using a tuple (Loc,Int), shortened to HndP.

$$\begin{aligned}\mathsf{Pool}\ \beta &\triangleq \textsc{PoolLeaf} \mid \textsc{PoolNode}\ (\mathsf{Pool}\ \beta)\ [\beta]\ (\mathsf{Pool}\ \beta) \\ \mathsf{HndP} &\triangleq (\mathsf{Loc},\mathsf{Int})\end{aligned}$$

Pools are manipulated with the following operations. These are all quite similar to the list operations and the individual lists within a pool are modified and queried using those actions.

$$\begin{aligned}\texttt{nextP} &:: \forall\beta.\ \mathsf{Loc} \to \mathsf{Pool}\ \beta \to \mathsf{Int} \\ \texttt{lkpP} &:: \forall\beta.\ \mathsf{HndP} \to \mathsf{Pool}\ \beta \to \beta \\ \texttt{ovwP} &:: \forall\beta.\ \mathsf{HndP} \to \beta \to \mathsf{Pool}\ \beta \to \mathsf{Pool}\ \beta \\ \texttt{newP} &:: \forall\beta.\ \mathsf{HndP} \to \beta \to \mathsf{Pool}\ \beta \to \mathsf{Pool}\ \beta \\ \texttt{stP} &:: \forall\beta.\ \forall\delta.\ (\beta \to (\beta,\delta)) \to \mathsf{HndP} \to (\mathsf{Pool}\ \beta \to (\mathsf{Pool}\ \beta, \delta))\end{aligned}$$

`nextP` l p returns the next free index in pool p at the list identified by location l. If `nextP` l $p = i$ then the handle `(`l`,`i`)` can be used later to refer to that index. `lkpP` and `ovwP` are look-up and overwrite operations for a particular element of a pool. As with the list operations, these fail if that element does not already exist. `newP` h b p creates a new structure in pool p of initial value b at h, and this only succeeds if h is actually the next free index. `stP` f h is a state-transformer on an entire pool which modifies the element identified by h using the state-transformer f and returns its value.

The definition of `newP` is complicated by the following question: a pool is a finite structure, so what happens if a particular location does not yet have a list associated with it when we want to allocate a new structure? The function `padP` solves this. `padP` l p modifies pool p by guaranteeing that for every location l_1 such that $l_1 \preccurlyeq l$, if there is not already a list associated with l_1 then a new empty list will be created.

$$\mathsf{padP} :: \forall\beta.\ \mathsf{Loc} \to \mathsf{Pool}\ \beta \to \mathsf{Pool}\ \beta$$

The following properties can be shown to hold. As per usual, many side-conditions relating to definedness are omitted.

Lemma 5.4.1.

1. *`lkpP` h (`ovwP` h v p) $= v$*
2. *`ovwP` h v_1 (`ovwP` h v_2 p) $=$ `ovwP` h v_1 p*
3. *If $h_1 \neq h_2$ then `lkpP` h_1 (`ovwP` h_2 v_2 p) $=$ `lkpP` h_1 p*
4. *If $h_1 \neq h_2$ then `ovwP` h_1 v_1 (`ovwP` h_2 v_2 p) $=$ `ovwP` h_2 v_2 (`ovwP` h_1 v_1 p)*
5. *If $h_1 \neq h_2$ then `newP` h_1 v_1 (`ovwP` h_2 v_2 p) $=$ `ovwP` h_2 v_2 (`newP` h_1 v_1 p)*
6. *If $l_1 \neq l_2$, `newP (`l_1`,`i_1`)` v_1 (`newP (`l_2`,`i_2`)` v_2 p) $=$ `newP (`l_2`,`i_2`)` v_2 (`newP (`l_1`,`i_1`)` v_1 p)*
7. *`nextP` l_1 (`ovwP` h_2 v_2 p) $=$ `nextP` l_1 p.*
8. *If $l_1 \neq l_2$ then `nextP` l_1 (`newP (`l_2`,`i_2`)` v_2 p) $=$ `nextP` l_1 p.*
9. *If $h_1 \neq h_2$ then* $\mathsf{cmmt}(\texttt{stP}\ f_1\ h_1, \texttt{stP}\ f_2\ h_2, p)$.
10. *If* $\mathsf{cmmt}(f_1, f_2, \texttt{lkpP}\ h\ p)$ *then* $\mathsf{cmmt}(\texttt{stL}\ f_1\ h, \texttt{stL}\ f_2\ h, p)$
11. *If $h_1 \neq h_2$ then* $\mathsf{cmmt}(\texttt{stP}\ f_1\ h_1, (\lambda w.\ \texttt{(newP}\ h_2\ v_2\ w\texttt{,}x\texttt{)}), p)$.

Proof. The proof for many of the above results is intricate. They involve induction over the tree structure of pools, induction over the list structure of locations and the use of all the results in Lemma 5.2.1. Since `newP` is defined using `padP`, we also need to prove a collection of separate lemmas that describe how `lkpP` and `ovwP` interact with `padP`. There are many

```
dmap′ w0 ⟨af′, wa′, ap′, pf′, ig′⟩ ≙ ⟨
    λ(l,la). λp. case la of
        DynAct h a →   applySnd Left (stP (af′ (l,a)) h p)
        Next →         (p,Right (l,nextP l p))
        Alloc (lh,ih) → if (l = lh) then ((newP (lh,ih) w0 p,Right ([],0))),
    λ(l,la). λp. case la of
        DynAct h a →   wa′ (l,a) (lkpP h p)
        Next →         False
        Alloc i →      False,
    λcm. λla. case la of
        DynAct h a →   apCxt ap′ (lkpM h cm) a
        Next →         True
        Alloc h →      isTopCxt (lkpM h cm),
    λl. λps. λcm. splitM (pf′ l) (λ(lh,ih). l⧺[L] ≼ lh) ps cm,
    λ(l,la). case la of
        DynAct h a →   ig′ a
        Next →         False
        Alloc h →      False
⟩
```

Figure 5.7: Definition of dmap′ combinator

of these, and they are omitted entirely, and are only accessible from the machine-readable proof sections (see Section B.3).

The difficulties are largely due to newP and padP. We have to prove that any nextP, lkpP and ovwP operations behave identically when attempting to access a location with no associated list, and a location with an associated empty list. □

5.4.2 Definition

The definition of the dmap′ combinator can found in Figure 5.7.

This definition is somewhat similar to the dmap definition found in Figure 5.3 and makes use of the same helper functions. Actions are elements of the type DynAction HndP α. Assuming that the current location is passed implicitly,

- DynAct h a performs action a on the structure identified by handle h, possibly stalling. This action is location-sensitive if and only if a is location-sensitive.
- Next returns the HndP of the next free handle. This never stalls and it is always location-sensitive (the implicit location must be a valid one for that process). The location part of the returned handle will be identical to that passed implicitly to it. It is the index which we need to obtain.
- Alloc h creates a brand new structure at h. This is both location-sensitive and, like with dmap, subject to the condition we have complete access to the structure at

handle h. There is also a small side-condition that the implicitly passed location and the location in h must be identical for the action to succeed. Because Alloc h will normally be performed straight after Next this should always hold.

When we split a context we must be careful when deciding what permissions are given by default to each side. If we are to allow multiple concurrent processes to allocate concurrently then these processes must have access to the future handles they may need. The default is that when splitting a context (l, c) into $(l{+\!\!+}[L], c_l)$ and $(l{+\!\!+}[R], c_r)$, the right-hand side by default gets access to all handles (l_h, i_h) except those where $l{+\!\!+}[L] \preccurlyeq l_h$. This means that under normal circumstances both sides will be able to allocate – that is, as long as one doesn't decide to distribute permissions for an as yet unallocated handle.

Theorem 5.4.1. *If* $\mathsf{PRE}'_{\mathfrak{s}}$ *then* $\mathsf{PRE}'_{dmap'\ w_0\ \mathfrak{s}}$.

Proof. Once again we apply Lemma 5.1.2 to show that the left- and right-hand sub-contexts have been split correctly for each index.

$c_l \sqsubseteq_{\texttt{dmap}'\ \mathfrak{s}} c$ and $c_r \sqsubseteq_{\texttt{dmap}'\ \mathfrak{s}} c$ are shown to hold in the same style as with the `smap` combinator. The main difficulty is in proving $c_l \lozenge'_{\texttt{dmap}'\ \mathfrak{s}} c_r$. There are three different types of actions and this means that there are six possible pairs of actions to account for. In each case the context and whether the actions are location-sensitive affects whether two actions can every be expected to run concurrently. For each action there is also the implicit location l_l and l_r, and we can assume that $l_l \neq l_r$ if both actions are location-sensitive.

- DynAct h_l a_l and DynAct h_r a_r. Similar to `dmap` in that we check if $h_l = h_r$. If $h_l \neq h_r$ then both actions affect different sections of the pool, and therefore commute. If $h_l = h_r$ then we must use $\mathsf{PRE}'_{\mathfrak{s}}$. Whether or not the actions are context sensitive carries over directly.

- Next and DynAct h_r a_r. These actions commute because Next does not change world state and DynAct h_r a_r does not change the length of any list within a pool. Since Next never changes world state it cannot cause DynAct h_r a_r to stall.

- Alloc h_l and DynAct h_r a_r. Like with `dmap` the proof of Alloc h_l $|||_{\texttt{dmap}'\ w_0\ \mathfrak{s}}$ DynAct h_r a_r arises from the fact that h_l cannot equal h_r, because for Alloc i_l to be permitted it must have complete access to handle h_l, thus forbidding any other actions. Once $h_l \neq h_r$ is known we use the fact that `newP` doesn't affect `stP`.

- Next and Next. Neither change world state so they both commute.

- Alloc h_l and Next. Both actions are location-sensitive so we know that the implicit locations l_l and l_r are not equal to each other. Therefore `newP` and `nextP` do not affect one another and the actions commute.

- Alloc h_l and Alloc h_r. Both actions are location-sensitive, so the two `newP` calls will allocate in different locations, and therefore be order independent.

□

The helper functions given for `dmap` in Figure 5.4 are almost identical when using `dmap′`. The main difference is that after converting a location-based model `dmap′` w_0 `s` back to a normal I/O model using `LMtoM` we must replace `action` and `test` with `actionL` and `testL`. These are used to pass the implicit `Loc` argument to each action.

5.5 Chapter summary

We have presented, in this chapter, a collection of useful, stock techniques for building larger, more sophisticated I/O models from simpler, smaller ones. Location-based I/O models were developed in order to solve some technical problems related to I/O actions which allocate structures dynamically. Through the use of location-based models, an action can now determine the identity of the process which calls it. Three basic combinators were developed – $*$, `smap` and `dmap` – as well as their more sophisticated location-based equivalents, $*'$, `smap′` and `dmap′`.

In Chapter 6 these are all put to good use, as we build a model of Haskell's I/O interface.

- The `dmap′` combinator is used to let the user allocate communication channels.
- The `smap′` combinator is used to construct the file system.
- The file model itself makes direct use of location-based facilities – the act of opening a file for reading requires one to allocate a new read handle.
- All the individual parts of the API are joined using the $*'$ combinator.

Chapter 6

A real world API and applications

In this chapter we finally put the theory to work and attempt to give a formal model of a real API. As one can expect, this is not a precise task. We try to anchor our semantics to reality as much as possible by remaining dedicated to the exact Haskell 98 I/O interface. The important contribution of this chapter is a semantics for a large, useful subset of the built-in Haskell I/O actions, additional deterministic communication primitives, and some applications which demonstrate the power and flexibility of Curio.

We begin in Section 6.1 by discussing which aspects of Haskell's I/O interface we either cannot model or are uninterested in modelling. Section 6.2 presents an I/O model $\mathfrak{file}$ which models an individual file in the file system. Section 6.3 describes two deterministic communication primitives, $\mathfrak{chan}$ and $\mathfrak{qsem}$. Section 6.4 then combines these smaller models using the combinators defined in Chapter 5 to give a full, real world I/O model $\mathfrak{io}$, and we provide libraries which mimic that of the Haskell API. Section 6.5 gives a few example applications.

6.1 Dissecting Haskell's I/O API

6.1.1 I/O primitives in Haskell 98

Our starting point is the standard libraries of Haskell 98, as defined in [90]. As well as providing a large collection of pure, non I/O-related libraries – such as for monads, arrays, complex arithmetic, and lists – there are also actual monadic I/O actions for accessing files, directories, doing terminal I/O, accessing the system clock etc. Naturally enough we are only concerned with those built-in functions which take place "within" the `IO` monad.

The libraries which contain `IO` actions are: `PreludeIO` (part of the main Haskell Prelude, containing very common I/O actions), `IO` (file access), `Directory` (directory access), `System` (calls such as `getArgs`, `system`, `exit`), `Time` and `CPUTime` (which allows us to retrieve the current clock time), and `Random` (a random number generator within the `IO` monad).

In general, we are concerned solely with modelling aspects of I/O which either:

- Give rise to opportunities for deterministic concurrency.
- May allow useful properties to be proved about them.

If, for some I/O action, neither of the above hold then we must ask what value there would be in giving a semantics to that action.

The actions which we do *not* consider interesting are:

- I/O actions which are merely **convenient wrappers** for other, predefined actions. This includes: `hPrint`, `readLn`, which make use of the `Show` and `Read` type classes and pre-process any text-based I/O; `hReady`, defined in terms of `hWaitForInput`; `putChar`, `getChar` and `isEOF`, which can be defined in terms of `hPutChar`, `hGetChar` and `hIsEOF`*; `writeFile` and `appendFile` which, although defined as primitive in `PreludeIO`, may be written just as well using other file operations.
- Semantically transparent actions which are solely related to **efficiency**. This includes: file buffering operations such as `getBuffering`, `setBuffering`, `hFlush`; the `renameFile` action which is semantically no different to a program which copies a file and then deletes the original.
- Actions related to **exception handling** or **errors**, such as `catch`, `ioError`, and the collection of actions which allow one to examine an IOError. We don't consider error handling to be a critical part of modelling I/O since a great many error/exceptions can be predicted in advance by checking EOF conditions, whether a file exists etc.
- System-dependent calls.
- **Lazy file I/O**. One of the most convenient features of the Haskell's file API is its use of lazy file I/O. The action

  `hGetContents :: Handle → IO String`

  reads the contents of a file lazily into a String. This means that the action terminates immediately, supposedly returning a string containing the complete file (or handle) contents. However the data is only read from disk when the contents of this string are queried, and in the meantime the file is in an intermediate "semi-closed" state. While convenient, this comes at a price:

  > "The contents of this final list is only partially specified: it will contain at least all the items of the stream that were evaluated prior to the handle becoming closed." [90]

*Actually, `getChar`, `putChar` and a few other actions related to standard input/output are not defined using `hGetChar` and `hPutChar` in Haskell since the former appear in `PreludeIO` and the latter in `IO`. But this is irrelevant since it is only done to ensure that one can do terminal I/O without having to import the whole `IO` module.

A subsequent `hClose` may close this handle and it is possible that parts of the string have not yet been evaluated. In the presence of lazy evaluation this may hard to predict, and the Haskell runtime system cannot guarantee that the resultant list will contain the entire file contents. (The actions `getContents` and `readFile` use the same mechanism).

We do not try to give a semantics for `hGetContents` not because its behaviour is uninteresting – quite the opposite – but since to do so properly would require a full *operational* semantics of lazy evaluation (i.e. Launchbury's [65]) and this is beyond the current capabilities of Curio.

- **Directory operations.** The `Directory` library contains several actions which let the user create, delete and query the contents of directories. As far as we are concerned, directories are merely a convenient form of organising individual files – a form of bookkeeping – and therefore do not exhibit many opportunities for concurrency beyond that of files themselves. Nor do they have many interesting semantic properties in their own right.

- **Temporal/time-related actions**, such as `getModificationTime`, `getClockTime` or `getCPUTime`. The simple reason for excluding these actions is that they are of limited use and where they *are* of use there is little of interest we can say about them in the semantics. One would need a useful way of modelling the fact that each primitive action takes a non-zero amount of time. This might not be too difficult. One can imagine a combinator which takes an I/O model and creates a new one in which there is a time counter and an extra action which lets one query the current time. Yet an action which queried the time could not then be performed concurrently with *any* other action, since that would cause non-determinism.

  Perhaps a more useful action would be one which waited until the current time counter exceeded some specific fixed timestamp. But the question remains: why would you do this, and if you did, what properties could you expect then to be able to prove about your program?

  The problem is that although temporal aspects of I/O are important, they come in three general varieties:

  1. Requirements that something takes place after something else, probably in separate processes. These relative time constraints do not need a global clock and basic interprocess communication primitives would suffice for this task.
  2. Interactive reasons: we want a program to display the modification time of a file, or write the current clock time to a file, or perhaps wait a fixed number of milliseconds so as to get a more desirable user interaction. It is hard to see how we could give a useful semantics to this.

3. Efficiency: for example we want a process to perform a low-priority task such as logging once every minute or two, so we must deliberately pause a process. As mentioned, these efficiency concerns are semantically transparent and not something we are interested in modelling.

For the above reasons, we do not attempt to model temporal properties of an I/O model.

- **System or OS** information. The effects of actions such as `getArgs`, `system`, `exit` or `getCurrentDirectory` are not worth trying to model, and similarly we don't try to give a semantics to the `stderr` handle.

- **Random Number** actions. Although not really related to I/O, the `Random` library defines a collection of standard random number libraries within the `IO` monad because it is convenient. Random numbers are difficult for us to model. In one sense reading a random number is an entirely commutative action (as mentioned in [87]), because the randomness of a number is not affected by the numbers which have gone before it. It should therefore be a perfect candidate for concurrency.

  But in another sense surely every program which relies on randomness is inherently non-deterministic? If we were to permit a single random number generator action to be called by multiple processes then our semantics would have to "know" that the actual value returned by this would be somehow irrelevant.

  Another possibility would be to use location-based models to guarantee that each process has its own local random number generator. This would work. Our programs would remain deterministic in the same sense that our model of terminal I/O is deterministic. That is: although we cannot really predict what will happen when we call the action, that same, unpredictable thing would have always occurred.

  Despite this, since random number generation is only in the `IO` monad for convenience, we choose to ignore it altogether.

Having removed many actions which are either uninteresting or difficult from a semantic point of view, we now have a core collection of I/O actions and structures which we will try to model. These can be found in Figure 6.1. The types are all identical to those given in the Haskell Report, with the exception of `hWaitForInput`. This function, which waits for input on a specific handle, normally contains a time-out mechanism whereby if after a set length of time no input has appeared on the handle then it returns. Since we are not modelling temporal properties of I/O, we force `hWaitForInput` to wait indefinitely.

6.1.2 Communications primitives

Concurrency primitives will be necessary if we are to write interesting programs which contain processes which communicate with one another. However, these cannot be normal

$$
\begin{array}{rcl}
\mathsf{FilePath} & \triangleq & \mathsf{String} \\
\mathsf{SeekMode} & \triangleq & \textsc{AbsoluteSeek} \mid \textsc{RelativeSeek} \mid \textsc{SeekFromEnd} \\
\mathsf{IOMode} & \triangleq & \textsc{ReadMode} \mid \textsc{WriteMode} \\
 & \mid & \textsc{AppendMode} \mid \textsc{ReadWriteMode} \\
 & & \\
\mathsf{Handle} & \triangleq & \cdots \\
\mathsf{HandlePosn} & \triangleq & \cdots \\
 & & \\
\texttt{stdin}, \texttt{stdout} & :: & \mathsf{Handle} \\
\texttt{openFile} & :: & \mathsf{FilePath} \to \mathsf{IOMode} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Handle} \\
\texttt{hClose} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ () \\
\texttt{hFileSize} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Int} \\
\texttt{hIsEOF} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Bool} \\
\texttt{hGetPosn} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{HandlePosn} \\
\texttt{hSetPosn} & :: & \mathsf{HandlePosn} \to \mathsf{Prog}_{\mathsf{io}}\ () \\
\texttt{hSeek} & :: & \mathsf{Handle} \to \mathsf{SeekMode} \to \mathsf{Int} \to \mathsf{Prog}_{\mathsf{io}}\ () \\
\texttt{hWaitForInput} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ () \\
\texttt{hGetChar}, \texttt{hLookAhead} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Char} \\
\texttt{hPutChar} & :: & \mathsf{Handle} \to \mathsf{Char} \to \mathsf{Prog}_{\mathsf{io}}\ () \\
\texttt{hIsOpen}, \texttt{hIsClosed} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Bool} \\
\texttt{hIsReadable}, \texttt{hIsWritable} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Bool} \\
\texttt{hIsSeekable} & :: & \mathsf{Handle} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Bool} \\
\texttt{removeFile} & :: & \mathsf{FilePath} \to \mathsf{Prog}_{\mathsf{io}}\ () \\
\texttt{doesFileExist} & :: & \mathsf{FilePath} \to \mathsf{Prog}_{\mathsf{io}}\ \mathsf{Bool}
\end{array}
$$

Figure 6.1: Chosen Haskell 98 I/O actions

concurrency actions because we must ensure that they do not admit non-determinism.

It is surprisingly hard to find details of general-purpose deterministic concurrency primitives in the literature. The dataflow language Id used "I-structures" [8], a kind of shared variable, to concurrently compute matrices in a deterministic, functional style. I-structures were modelled in Chapter 2 but they are extremely weak since they may only ever be written to once. We instead look to Concurrent Haskell to see what operations we could salvage.

Concurrent Haskell [89] describes the concurrent, non-deterministic extensions to the Haskell language. The Revised Haskell Report does not describe these new communication primitives but both the Glasgow Haskell Compiler [34] and Hugs [55] implement them. All communication in Concurrent Haskell is implemented using `MVar`s, mutable variables which allow synchronisation between processes – if a process tries to obtain an empty `MVar` then it will block until it has been written to. The other more high-level communication tools are `Chan`s (unbounded FIFO channels) and `QSem`s (quantity semaphores).

In order to ensure determinism we must never allow actions to compete for a limited resource. This means that any `MVar`-like synchronisation variable implemented as primitive could have only one reader and one writer. Therefore, writing a stream of data to a reading process would have to be done in lock-step. While it may be possible to implement communication channels this way, it would appear to be a bit low-level for our needs. Instead we use channels themselves as our primary means of letting processes communicate. This gives a more satisfactory interface, because we already access files and terminal I/O via handles which allow the reading and writing of streams of data. By defining a handle structure which at a high-level also accommodates communication channels we can unify our high-level interface to both file I/O and interprocess communication.

There are two specific sorts of communication channel which we want to permit. These can both be encapsulated within the read/write file handle framework:

- One-to-one FIFO communication channels: one process sends, the other receives. This example was given in Chapter 2, but we will enhance it slightly to allow for some extra operations. If one needs a one-to-many communication buffer in which the sent data is duplicated to each individual writer then this can be emulated with a collection of one-to-one channels. These channels will only be able to communicate `Char`s.

- Many-to-one quantity semaphores: these are necessary because in practice multiple processes may communicate with a single one deterministically just as long as those individual sender processes cannot actually be distinguished. This is really just a quantity semaphore, where the multiple "sender" processes increment a counter while the "receive" process decrements the counter, blocking if the counter is zero. Thus the receiving process cannot tell the many sending processes apart. This is a somewhat weak primitive but it is included because it cannot be emulated with a collection of one-to-one buffers – any primitive which allowed a process to wait simultaneously for the first input on two different channels would then immediately be non-deterministic.

6.1.3 Extending the Haskell interface

Since we propose to incorporate communication channels and quantity semaphores within the file-handle framework, and because I/O contexts mean that certain actions may not be permitted, we must provide a few more library functions in addition to those defined in the Haskell 98 report. These are:

$$\begin{aligned}
\texttt{newChannel}, \texttt{newQSem} &:: \mathsf{Prog}_{\mathfrak{io}}\ (\mathsf{Handle}, \mathsf{Handle}) \\
\texttt{fIsOpen} &:: \mathsf{FilePath} \to \mathsf{Prog}_{\mathfrak{io}}\ \mathsf{Bool} \\
\texttt{hAllowed} &:: \mathsf{Handle} \to \mathsf{Prog}_{\mathfrak{io}}\ \mathsf{Bool} \\
\texttt{fAllowedR}, \texttt{fAllowedW} &:: \mathsf{FilePath} \to \mathsf{Prog}_{\mathfrak{io}}\ \mathsf{Bool} \\
\texttt{parIO} &:: [\mathsf{Handle}] \to \mathsf{Prog}_{\mathfrak{io}}\ \beta \to [\mathsf{Handle}] \to \mathsf{Prog}_{\mathfrak{io}}\ \gamma \to \mathsf{Prog}_{\mathfrak{io}}\ (\beta, \gamma)
\end{aligned}$$

`newChannel` is used to dynamically allocate a new communication channel, returning its separate read and write handles. `newQSem` behaves in the same way for quantity semaphores. `fIsOpen` returns whether a file is currently opened, either for reading or writing. The function `hAllowed` is used to check to see if the current I/O context allows one to use a particular handle. `fAllowedW` and `fAllowedR` check to see if the current context has write-access (i.e. complete access) or read access to a particular file. The behaviour of these three actions is entirely orthogonal to that of the actions `isReadable` and `isWritable`, which merely query information about the handle itself. To be completely safe, when, say, opening a file for reading, we must check first whether the action is permitted by the I/O context, and then also check to see if the action, though allowed, will not cause an error (i.e. if the file is already open for writing, or if the file doesn't exist). Lastly, $\mathfrak{parIO}$ is a convenient wrapper for Curio's `par`, allowing one to perform concurrency by specifying the handles we wish each sub-process to have access to.

A handle is admittedly a rather crude interface to a quantity semaphore – one signals using `hPutChar` and one waits using `hGetChar`, but no data is actually transferred. There is a great advantage, however, to accessing all I/O resources via a single data type: when performing concurrency it allows us to unify the way in which we allocate permissions to different processes, as we shall show later.

6.2 Modelling an individual file

Our primary concern when modelling real world I/O is describing a file system. In this section we define an I/O model $\mathfrak{file}$ which forms the cornerstone of our file system model. This model will be far removed from the elegant Haskell 98 interface – it will be shown later how the interface to these actions can be tidied up.

We want the following basic properties to hold for each individual file:

- A file may or may not exist, and if it exists it should contain a finite sequences of

characters (the file contents).

- If a file exists it should also contain information concerning whether it is open for (possibly shared) reading or (exclusive) writing, appending or reading/writing, and in both cases contain a structure which maintains knowledge of the various active file-pointers.
- Two concurrent processes should be able to simultaneously perform actions which manipulate two different read handles opened on a single file.
- Two processes should ideally also be able to simultaneously open a file for shared reading – that is, the creation of a new read handle should not be order dependent.
- We should be able to perform all the actions defined in Figure 6.1.

There have been a few other attempts to develop a semantic model of a file system with a view to proving correctness properties [7, 125, 14]. None of these try to give an adequate interpretation of what it means for the opening of two read handles to be order independent, so we developed our own approach ([14] probably comes closest, but its read handles for a given file cannot be distinguished, meaning that a file handle may be closed twice).

6.2.1 The file world state and I/O contexts

How should the internal file-state be represented? The file data can be stored as a list of characters, [Char], and if the file is open for writing then the single write pointer can be stored as an integer.

The most difficult obstacle to overcome is how we should permit two concurrent processes to allocate, modify and close two distinct read pointers. The main question is: what structure should be used to store these pointers, and how does that structure guarantee that two processes can create new read pointers in an order independent fashion? Also there must be some notion of a closed file-pointer. How do we ensure that the closing of a file-pointer in one process does not affect the creation of a new file-pointer in another concurrent process, causing that process to possibly obtain a different pointer? This also applies to the closing of an entire file. Presumably a file becomes closed once each file-pointer which was in use becomes closed. If a file is closed, and then two processes concurrently open a new file-pointer, read from it and then close it, we must also guarantee that when a file is closed that we don't "reset" our structure for storing read pointers.

We solve this in the following way:

- A Pool FPtr is used to store all the individual read pointers[†]. This guarantees that we can allocate and modify two separate read pointers concurrently.

[†]Pools were defined in Section 5.4.

- A FPtr denotes either an active read pointer or one which has been closed. This makes sure that even after a read pointer has been closed it is still present, somehow, and cannot be reused.
- Instead of having a separate tag which indicates if the file itself is closed, we just *define* a file to be closed when all read pointers are themselves closed.

Since we use pools it makes sense to make the model location-based. So $\mathfrak{file}$ has the following overall structure:

$$\begin{aligned} \mathfrak{file} &:: \mathsf{IOModelL}\ \nu_{\mathtt{File}}\ \alpha_{\mathtt{File}}\ \rho_{\mathtt{File}}\ \omega_{\mathtt{File}}\ \varsigma_{\mathtt{File}} \\ \mathfrak{file} &\triangleq \langle \mathtt{af}'_{\mathtt{File}}, \mathtt{wa}'_{\mathtt{File}}, \mathtt{ap}'_{\mathtt{File}}, \mathtt{pf}'_{\mathtt{File}}, \mathtt{ig}'_{\mathtt{File}} \rangle \end{aligned}$$

The world state of a file, $\omega_{\mathtt{File}}$ is defined as:

$$\begin{aligned} \omega_{\mathtt{File}} &\triangleq \textsc{NoFile} \mid \textsc{File}\ [\mathsf{Char}]\ (\textsc{WrPtr}\ \mathsf{Int}\ \mathsf{IOMode} \mid \textsc{RdPtrs}\ (\mathsf{Pool}\ \mathsf{FPtr})) \\ \mathsf{FPtr} &\triangleq \textsc{FPtr}\ \mathsf{Int} \mid \textsc{Stale} \end{aligned}$$

If a file doesn't exist it is NoFile. If it does exist then it is either File cs (WrPtr i m), where i is the single write pointer and m represents the mode in which the file was opened, or File cs (RdPtrs p), where p is the pool of read pointers, some of which may be closed, or Stale. In both cases cs is the file data.

Each FPtr within the pool is identified using a handle h :: HndP. When performing concurrency we must guarantee that different processes accessing the same file have access to different read pointers. This gives rise to the following I/O context for files:

$$\varsigma_{\mathtt{File}} \triangleq \textsc{FTotal} \mid \textsc{FReadAccess}\ (\mathsf{HndP} \rightarrow \mathsf{Bool}) \mid \textsc{FNone}$$

A program may either have complete access to a particular file, FTotal, it may have access to a particular set of read handles, FReadAccess m, or it may have no access to the file at all, FNone. (There are approaches to allowing concurrency on a single file which are more fine-grained. As an example, if a file is open for writing but is not allowed to reopen that file, thereby truncating it, then we may allow another concurrent process to wait until that file has exceeded a certain length. We will not consider these here).

One splits I/O context $\varsigma_{\mathtt{File}}$ using the following type:

$$\rho_{\mathtt{File}} \triangleq [(\mathsf{HndP}, \mathsf{Dir})]$$

$\mathtt{pf}'_{\mathtt{File}}$ is defined in a similar fashion to that of the $\mathtt{dmap}'$ combinator. We omit the definition, but supplied with parameters l :: Loc, p :: $\rho_{\mathtt{File}}$ and c :: $\varsigma_{\mathtt{File}}$ it behaves as follows:

- If c is FNone then both sub-contexts become FNone.

- If c is FReadAccess m then $p :: \rho_{\texttt{File}}$, the list of `(HndP,Dir)`, dictates whether the left-hand or right-hand side gets access to the various read handles originally accessible. By default a handle (l_h, i_h) goes to the left-hand-side if and only if $l{+\!\!+}[\mathrm{L}] \preccurlyeq l_h$.
- If c is FTotal and $p = []$ then the resultant left- and right-hand contexts become FNone and FTotal. Otherwise it behaves as if $c =$ FReadAccess m_0, where m_0 indicates that all handles may be accessed.

6.2.2 File actions

Next we must isolate those actions which only affect a single read pointer from the actions which affect the entire file. It was found that there were two general varieties of actions on a file, and four other anomalous actions which had to be treated entirely separately. This means that at the top-level there are just six actions, and the action type $\alpha_{\texttt{File}}$ is:

$$
\begin{aligned}
\alpha_{\texttt{File}} \triangleq\ & \textsc{WholeFileAct}\ \mathsf{WholeFileAction} \\
\mid\ & \textsc{FilePtrAct}\ (\textsc{WPtr} \mid \textsc{RPtr}\ \mathsf{HndP})\ \mathsf{FilePtrAction} \\
\mid\ & \textsc{FDoesExist} \mid \textsc{FNextRdPtr} \mid \textsc{FRdOpen}\ \mathsf{HndP} \mid \textsc{FRdHIsOpen}\ \mathsf{HndP}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{WholeFileAction} \triangleq\ & \textsc{FRemove} \mid \textsc{FPutChar}\ \mathsf{Char} \mid \textsc{HWrOpen}\ \mathsf{IOMode} \\
\mid\ & \textsc{FWrIsOpen} \mid \textsc{FIsOpen}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{FilePtrAction} \triangleq\ & \textsc{FGetChar} \mid \textsc{FIsEOF} \mid \textsc{FSeek}\ \mathsf{SeekMode}\ \mathsf{Int} \\
\mid\ & \textsc{FLookAhead} \mid \textsc{FClosePtr} \mid \textsc{FileSize} \mid \textsc{FGetPosn}
\end{aligned}
$$

The two general schemas for file actions are

- WholeFileAct: actions which affect the entire file and must be explicitly sequenced with respect to all other actions on that file. These are:
  - FRemove: delete the file.
  - FPutChar: write a character to the file, failing if it is not open for writing.
  - HWrOpen: open the file for writing in a particular mode, failing if it it not closed.
  - FWrIsOpen: return whether the file is open for writing.
  - FIsOpen: return whether there are any active read or write pointers.
- FilePtrAct: actions which only affect an existing write pointer or read pointer, and query, but do not change, the file contents. These actions are:

- FGetChar: read the next character from the file, incrementing the pointer. This fails if the pointer is invalid.
- FIsEOF: return whether the pointer points to the end-of-file. This returns true if the pointer equals the length of the file data.
- FSeek: change the current pointer.
- FLookAhead: read the next character without incrementing the pointer.
- FClosePtr: close the pointer. Closing an already closed pointer is not an error.
- FileSize: return the length of the file data.
- FGetPosn: return the current pointer position.

The anomalous actions are:

- FNextRdPtr and FRdOpen, which together open a file for reading. Firstly one determines the next free handle in the pool; then one creates a new read pointer at that handle. These two actions are location-sensitive – and they are the only location-sensitive actions in the file model.
- FRdHIsOpen: check if the pointer associated with a read handle is open. This is a meta-pointer operation. Unlike all the other actions that modify pointers we do not want this to fail if the pointer is closed or not yet in existence.
- FDoesExist: check if a file exists. We could have classified this as an action which must be single threaded (like deleting a file, or writing to a file), but this would be overkill. If many processes have only read access to a file then they cannot either delete or create that file, so those processes can also query whether the file exists. It can therefore be run by a process if it has access to at least one read handle, because this guarantees that there will be no writing process in parallel.

This is formalized in Figure 6.2. The semantics of the actions actions which fall under one of the two schemas are encoded with `doWFA` and `doFPA`. These function are defined in Appendix A.3. The definition is of no relevance to the confluence proof for file – the interface by which the actions are defined guarantees all the necessary properties (i.e. actions which affect only a single read pointer are incapable of changing the file contents).

The type of return values, $\nu_{\texttt{File}}$, is just a sum-type which lets one store whatever particular values file actions may return.

$$\nu_{\texttt{File}} \triangleq \text{RNull} \mid \text{RBool Bool} \mid \text{RHndP HndP} \mid \text{RInt Int} \mid \text{RChar Char}$$

6.2.3 Confluence of file model

We now prove that the I/O model file is confluent.

$$
\begin{aligned}
\mathtt{ig}'_{\mathtt{File}} &:: \alpha_{\mathtt{File}} \to \mathsf{Bool}\\
\mathtt{af}'_{\mathtt{File}} &:: (\mathsf{Loc}, \alpha_{\mathtt{File}}) \to \omega_{\mathtt{File}} \to (\omega_{\mathtt{File}}, \nu_{\mathtt{File}})\\
\mathtt{wa}'_{\mathtt{File}} &:: (\mathsf{Loc}, \alpha_{\mathtt{File}}) \to \omega_{\mathtt{File}} \to \mathsf{Bool}\\
\\
\mathtt{doWFA} &:: \mathsf{WholeFileAction} \to \omega_{\mathtt{File}} \to (\omega_{\mathtt{File}}, \nu_{\mathtt{File}})\\
\mathtt{doFPA} &:: \mathsf{FilePtrAction} \to [\mathsf{Char}] \to \mathsf{Int} \to (\mathsf{FPtr}, \nu_{\mathtt{File}})
\end{aligned}
$$

$\mathtt{af}'_{\mathtt{File}}$ (_, WholeFileAct a) w = doWFA a w
$\mathtt{af}'_{\mathtt{File}}$ (_, FilePtrAct WPtr a) (File cs (WrPtr i m)) = case (doFPA a cs i) of
  (FPtr i_1, v) →(File cs (WrPtr i_1 m), v)
  (Stale, v) → (File cs (RdPtrs PoolLeaf), v)
$\mathtt{af}'_{\mathtt{File}}$ (_, FilePtrAct (RdPtr h) a) (File cs (RdPtrs p)) = case (lkpP h p) of
  FPtr i → case (doFPA a cs i) of
    (f, v) → (File cs (RdPtrs (ovwP h f p)), v)
$\mathtt{af}'_{\mathtt{File}}$ (_, FDoesExist) w = constST (λw.case w of
  NoFile → RBool False
  File _ _ → RBool True)
$\mathtt{af}'_{\mathtt{File}}$ (l, FNextRdPtr) w = constST (λ(File cs (RdPtrs p)). RHndP (l, nextP l p))
$\mathtt{af}'_{\mathtt{File}}$ (l, FRdOpen (l_h, i_h)) (File cs (RdPtrs p)) =
  if ($l = l_h$) then (File cs (RdPtrs (newP (l_h, i_h) (FPtr 0) p)), RNull)
$\mathtt{af}'_{\mathtt{File}}$ (_, FRdHIsOpen h) w = constST (λw.case w of
  File cs (RdPtrs p) → case (lkpP h p) of
    FPtr i →RBool True
    Stale → RBool False
  _ → RBool False)

$$
\begin{aligned}
\mathtt{ap}'_{\mathtt{File}}\ \ &\mathrm{FTotal} && a &= \mathrm{True}\\
\mathtt{ap}'_{\mathtt{File}}\ \ &\mathrm{FNone} && a &= \mathrm{False}\\
\mathtt{ap}'_{\mathtt{File}}\ \ &(\mathrm{FReadAccess}\ m) && a &= \begin{cases}
\mathrm{False} & , a = \mathrm{WholeFileAct}\ x\\
\mathrm{False} & , a = \mathrm{FilePtrAct\ WrPtr}\ x\\
\mathtt{lkpM}\ h\ m & , a = \mathrm{FilePtrAct\ (RdPtr}\ h)\ x\\
\mathtt{lkpM}\ h\ m & , a = \mathrm{FRdOpen}\ h\\
\mathtt{lkpM}\ h\ m & , a = \mathrm{FRdHIsOpen}\ h\\
\mathrm{True} & , a = \mathrm{FDoesExist}\\
\mathrm{True} & , a = \mathrm{FNextRdPtr}
\end{cases}
\end{aligned}
$$

```
constST :: ∀β. ∀γ. (β → γ) → β → (β,γ)
constST f s = (s, f s)
```

$\mathtt{wa}'_{\mathtt{File}}$ (l, a) $w \triangleq$ False

$$
\mathtt{ig}'_{\mathtt{File}}\ a \triangleq \begin{cases}
\mathrm{False} & , a = \mathrm{FNextRdPtr}, a = \mathrm{FRdOpen}\ h\\
\mathrm{True} & , \text{otherwise}
\end{cases}
$$

Figure 6.2: High-level semantics of file actions

Theorem 6.2.1. $\mathsf{PRE}'_{\mathfrak{file}}$.

Proof. Assuming $\mathtt{pf}'_{\mathtt{File}}\ l\ p\ c = (c_l, c_r)$ it must be proved that $c_l \sqsubseteq_{\mathfrak{file}} c$, $c_r \sqsubseteq_{\mathfrak{file}} c$ and $c_l \lozenge'_{\mathfrak{file}} c_r$.

If $c_l = \textsc{FNone}$ and $c_r = \textsc{FTotal}$ then this is not difficult. The context c must equal FTotal. FNone $\sqsubseteq_{\mathfrak{file}}$ FTotal and FNone $\lozenge'_{\mathfrak{file}}$ FTotal both hold because context FNone does not permit any actions at all. FTotal $\sqsubseteq_{\mathfrak{file}}$ FTotal is true by reflexivity.

Otherwise $c_l = \textsc{FReadAccess}\ m_l$ and $c_r = \textsc{FReadAccess}\ m_r$, and for any h, it is impossible that both $\mathtt{lkpM}\ h\ m_l$ and $\mathtt{lkpM}\ h\ m_r$ are True (for any given read pointer no more than one of the two contexts will permit access to it). $c_l \sqsubseteq_{\mathfrak{file}} c$ and $c_r \sqsubseteq_{\mathfrak{file}} c$ are proved using the knowledge that c is either FTotal which permits all actions or it is FReadAccess m, where for each individual handle h, m_l and m_r cannot permit access to a read pointer if m forbids it.

The main proof obligation now becomes FReadAccess $m_l \lozenge'_{\mathfrak{file}}$ FReadAccess m_r. Since no action is ever stalled in the file model the $\mathsf{ally}_{\mathfrak{file}}$ condition always holds trivially. This means we just need to show that for all actions a_l, a_r permitted by contexts FReadAccess m_l and FReadAccess m_r respectively, $a_l \mid\mid\mid_{\mathfrak{file}} a_r$. There are six types of action (the two schemas and the four anomalous actions), so this would immediately suggest 21 different possible pairs of actions, up to symmetry, which need to be proved for. These can be pruned down quite quickly:

- Action of the form WholeFileAct a or FilePtrAct WPtr a can only be run in context FTotal, so can be excluded entirely.
- Actions FDoesExist, FRdHIsOpen h and FNextRdPtr do not change world state, so all of these actions automatically commute with one another.
- The only actions which create or delete the file are of the form WholeFileAct a, and these are excluded. Therefore the action FDoesExist commutes with all relevant actions.

There are seven remaining pairs of actions. From the way contexts are split we know that if two actions have associated handles h_l and h_r then $h_l \neq h_r$.

1. FilePtrAct (RPtr h_l) $a'_l \mid\mid\mid_{\mathfrak{file}}$ FRdOpen h_r
2. FNextRdPtr $\mid\mid\mid_{\mathfrak{file}}$ FRdOpen h_r.
3. FRdOpen $h_l \mid\mid\mid_{\mathfrak{file}}$ FRdOpen h_r.
4. FilePtrAct (RPtr h_l) $a'_l \mid\mid\mid_{\mathfrak{file}}$ FNextRdPtr.
5. FilePtrAct (RPtr h_l) $a'_l \mid\mid\mid_{\mathfrak{file}}$ FilePtrAct (RPtr h_r) a'_r.
6. FilePtrAct (RPtr h_l) $a'_l \mid\mid\mid_{\mathfrak{file}}$ FRdHIsOpen h_r.

7. FRdOpen h_l $|||_{\mathfrak{file}}$ FRdHIsOpen h_r.

For the first five the proofs are very similar to that of the `dmap′` combinator – actions FilePtrAct (RPtr h) a, FNextRdPtr and FRdOpen h closely correspond to the actions DynAct h a, Next and Alloc h used with `dmap′`. For the final two pairs of actions we use the fact that $h_l \neq h_r$. Neither FilePtrAct (RPtr h_l) a'_l nor FRdOpen h_l can affect whether there is a read pointer at h_r or, if one does exist, if it is closed. □

6.3 Two communication primitives

We now develop two simple models which act as communication primitives:

- A one-to-one FIFO communication channel, $\mathfrak{chan}$. This is slightly more complex than the $\mathfrak{bffr}$ example given in Chapter 2 in that it allows the channel to be closed by the sender, signalling that no more communication will take place. It also allows one to observe the next character from the channel without consuming it. These changes are made so as to make channels fit Haskell's file-handle interface more neatly.
- A quantity semaphore, $\mathfrak{qsem}$. Many concurrent processes may increase an integer counter, and one single process may decrease it, stalling if the counter is zero or negative.

We will develop both of these as normal I/O models, without location information, converting them afterwards to location-based models using `MtoLM`.

6.3.1 Communication channels

The I/O model $\mathfrak{chan}$ is defined in full in Figure 6.3. There are five actions:

- ChSend c: send character c.
- ChClose: close the channel permitting no further sending.
- ChIsClosed: check if the channel is closed.
- ChRcve: read a character if one is present. If no character is present and the channel is closed then return immediately. Otherwise wait until either a character appears or the channel is closed.
- ChLook: the same as ChRcve except it does not consume the character.

The I/O context for channels lumps the five actions into two groups: those associated with sending, the first three above, and those associated with receiving, the final two. It is in this fashion that permissions are allocated to sub-programs. At most one process is allowed to perform ChSend, ChClose and ChIsClosed, and one (probably different) process is allowed to perform ChRcve and ChLook.

Theorem 6.3.1. $\mathsf{PRE}_{\mathfrak{chan}}$.

Proof. When a context $c = \textsc{ChCxt}\ b_s\ b_r$ is split into two sub-contexts the sub-contexts are $c_l = \textsc{ChCxt}\ (b_s\ \texttt{\&\&}\ \texttt{not}\ b'_s)\ (b_r \texttt{\&\&}\ \texttt{not}\ b'_r)$ and $c_r = \textsc{ChCxt}\ (b_s\ \texttt{\&\&}\ b'_s)\ (b_r \texttt{\&\&}\ b'_r)$.

That $c_l \sqsubseteq_{\mathfrak{chan}} c$ and $c_r \sqsubseteq_{\mathfrak{chan}} c$ hold is the result of the fact that $b_s\ \texttt{\&\&}\ \texttt{not}\ b'_s$ and $b_s\ \texttt{\&\&}\ b'_s$ both imply b_s, and similarly that $b_r\ \texttt{\&\&}\ \texttt{not}\ b'_r$ and $b_r\ \texttt{\&\&}\ b'_r$ both imply b_r.

To prove $c_l \lozenge_{\mathfrak{chan}} c_r$, the only permitted, non-trivial case is when one context permits sending and the other permits receiving. That is, $c_l = \textsc{ChCxt}\ \textsc{True}\ \textsc{False}$ and $c_r = \textsc{ChCxt}\ \textsc{False}\ \textsc{True}$ (or where c_l and c_r are swapped, for which the proof is identical).

Only ChLook and ChRcve can become stalled and this only happens when the channel is both empty and open. However, neither of the three other actions can either cause a non-empty channel to become empty or cause a closed channel to become open, so the $\mathsf{ally}_{\mathfrak{chan}}$ condition always holds.

There are six pairs of actions which we must prove to be order independent, but we ignore the three involving ChLook since its behaviour is so similar to ChRcve.

- ChRcve $|||_{\mathfrak{chan}}$ ChSend c. If the channel is closed then neither actions do anything. If it is open then, for the receive action to be non-stalled the channel must be non-empty. Therefore both actions modify different sections of the list of characters.
- ChRcve $|||_{\mathfrak{chan}}$ ChClose. If a character was available to receive then closing the channel will still cause the character to be read correctly. If no characters were available then the channel must have been closed (both actions must be non-stalled), and closing it again will have no effect.
- ChRcve $|||_{\mathfrak{chan}}$ ChIsClosed. Not difficult. ChIsClosed does not affect the list of characters and ChRcve does not affect whether the channel is closed.

□

6.3.2 Quantity semaphores

Figure 6.4 defines the I/O model $\mathfrak{qsem}$, a quantity semaphore. This is a very simple model. The world state is just an integer and there are two actions: SWait waits until the counter is positive and then decrements it and SSignal increments the counter. It is always possible to perform SSignal, but only one process is allowed to perform SWait. The I/O context is a Bool which indicates whether the process can perform SWait.

Theorem 6.3.2. $\mathsf{PRE}_{\mathfrak{qsem}}$.

Proof. A context c is split into two sub-contexts False and c – if c permits SWait then only one sub-process may inherit this capability. Depending on value of $p :: \rho_{\mathtt{QSem}}$, this may be given to the left- or right-hand sub-context, but since the behaviour is symmetric, let us assume that $c_l = \textsc{False}$ and $c_r = c$.

$$\mathfrak{chan} :: \mathsf{IOModel}\ \nu_{\mathtt{Chan}}\ \alpha_{\mathtt{Chan}}\ \rho_{\mathtt{Chan}}\ \omega_{\mathtt{Chan}}\ \varsigma_{\mathtt{Chan}} \triangleq \langle \mathtt{af}_{\mathtt{Chan}}, \mathtt{wa}_{\mathtt{Chan}}, \mathtt{ap}_{\mathtt{Chan}}, \mathtt{pf}_{\mathtt{Chan}} \rangle$$

$$\begin{array}{lcllcl}
\nu_{\mathtt{Chan}} & \triangleq & \mathsf{Maybe\ Char} & \alpha_{\mathtt{Chan}} & \triangleq & \textsc{ChSend}\ \mathsf{Char} \mid \textsc{ChClose} \mid \textsc{ChIsClosed} \\
\rho_{\mathtt{Chan}} & \triangleq & \textsc{ChSpl}\ \mathsf{Bool}\ \mathsf{Bool} & & \mid & \textsc{ChLook} \mid \textsc{ChRcve} \\
\varsigma_{\mathtt{Chan}} & \triangleq & \textsc{ChCxt}\ \mathsf{Bool}\ \mathsf{Bool} & \omega_{\mathtt{Chan}} & \triangleq & (\mathsf{[Char]}, \textsc{ChOpen} \mid \textsc{ChClosed})
\end{array}$$

$$\begin{array}{llll}
\mathtt{af}_{\mathtt{Chan}} & (\textsc{ChSend}\ c) & (cs, \textsc{ChClosed}) & \triangleq ((cs, \textsc{ChClosed}), \textsc{Nothing}) \\
\mathtt{af}_{\mathtt{Chan}} & (\textsc{ChSend}\ c) & (cs, \textsc{ChOpen}) & \triangleq ((cs +\!\!+ [c], \textsc{ChOpen}), \textsc{Just}\ c) \\
\mathtt{af}_{\mathtt{Chan}} & \textsc{ChClose} & (cs, oc) & \triangleq ((cs, \textsc{ChClosed}), \textsc{Nothing}) \\
\mathtt{af}_{\mathtt{Chan}} & \textsc{ChIsClosed} & (cs, oc) & \triangleq ((cs, oc), \mathtt{case}\ oc\ \mathtt{of} \\
 & & & \quad \textsc{ChOpen} \rightarrow \textsc{Just}\ 'O' \\
 & & & \quad \textsc{ChClosed} \rightarrow \textsc{Just}\ 'C') \\
\mathtt{af}_{\mathtt{Chan}} & \textsc{ChLook} & ((c : cs), oc) & \triangleq (((c : cs), oc), \textsc{Just}\ c) \\
\mathtt{af}_{\mathtt{Chan}} & \textsc{ChLook} & ([], \textsc{Closed}) & \triangleq (([], \textsc{Closed}), \textsc{Nothing}) \\
\mathtt{af}_{\mathtt{Chan}} & \textsc{ChRcve} & ((c : cs), oc) & \triangleq ((cs, oc), \textsc{Just}\ c) \\
\mathtt{af}_{\mathtt{Chan}} & \textsc{ChRcve} & ([], \textsc{Closed}) & \triangleq (([], \textsc{Closed}), \textsc{Nothing})
\end{array}$$

$$\begin{array}{llll}
\mathtt{wa}_{\mathtt{Chan}} & (\textsc{ChSend}\ c) & (cs, oc) & \triangleq \textsc{False} \\
\mathtt{wa}_{\mathtt{Chan}} & \textsc{ChClose} & (cs, oc) & \triangleq \textsc{False} \\
\mathtt{wa}_{\mathtt{Chan}} & \textsc{ChIsClosed} & (cs, oc) & \triangleq \textsc{False} \\
\mathtt{wa}_{\mathtt{Chan}} & \textsc{ChLook} & (cs, oc) & \triangleq \mathtt{null}\ cs\ \mathtt{\&\&}\ oc = \textsc{ChOpened} \\
\mathtt{wa}_{\mathtt{Chan}} & \textsc{ChRcve} & (cs, oc) & \triangleq \mathtt{null}\ cs\ \mathtt{\&\&}\ oc = \textsc{ChOpened}
\end{array}$$

$$\begin{array}{llll}
\mathtt{ap}_{\mathtt{Chan}} & (\textsc{ChCxt}\ b_s\ b_r) & (\textsc{ChSend}\ c) & \triangleq b_s \\
\mathtt{ap}_{\mathtt{Chan}} & (\textsc{ChCxt}\ b_s\ b_r) & \textsc{ChClose} & \triangleq b_s \\
\mathtt{ap}_{\mathtt{Chan}} & (\textsc{ChCxt}\ b_s\ b_r) & \textsc{ChIsClosed} & \triangleq b_s \\
\mathtt{ap}_{\mathtt{Chan}} & (\textsc{ChCxt}\ b_s\ b_r) & \textsc{ChLook} & \triangleq b_r \\
\mathtt{ap}_{\mathtt{Chan}} & (\textsc{ChCxt}\ b_s\ b_r) & \textsc{ChRcve} & \triangleq b_r
\end{array}$$

$$\mathtt{pf}_{\mathtt{Chan}}(\textsc{ChSpl}\ b'_s\ b'_r)(\textsc{ChCxt}\ b_s\ b_r) \triangleq$$
$$(\textsc{ChCxt}\ (b_s\ \mathtt{\&\&}\ \mathtt{not}\ b'_s)\ (b_r \mathtt{\&\&}\ \mathtt{not}\ b'_r), \textsc{ChCxt}\ (b_s\ \mathtt{\&\&}\ b'_s)\ (b_r\ \mathtt{\&\&}\ b'_r))$$

Figure 6.3: $\mathfrak{chan}$ – a one-to-one communication channel

$$\mathfrak{qsem} :: \mathsf{IOModel}\ \nu_{\mathtt{QSem}}\ \alpha_{\mathtt{QSem}}\ \rho_{\mathtt{QSem}}\ \omega_{\mathtt{QSem}}\ \varsigma_{\mathtt{QSem}} \triangleq \langle \mathtt{af}_{\mathtt{QSem}}, \mathtt{wa}_{\mathtt{QSem}}, \mathtt{ap}_{\mathtt{QSem}}, \mathtt{pf}s_{\mathtt{QSem}} \rangle$$

$$\begin{array}{lcllcl}
\nu_{\mathtt{QSem}} & \triangleq & () & \omega_{\mathtt{QSem}} & \triangleq & \mathsf{Int} \\
\rho_{\mathtt{QSem}} & \triangleq & \mathrm{L} \mid \mathrm{R} & \varsigma_{\mathtt{QSem}} & \triangleq & \mathsf{Bool}
\end{array} \qquad \alpha_{\mathtt{QSem}} \triangleq \textsc{SWait} \mid \textsc{SSignal}$$

$$\begin{array}{llllll}
\mathtt{af}_{\mathtt{QSem}} & \textsc{SWait} & i & \triangleq (i-1, ()) & \mathtt{wa}_{\mathtt{QSem}} \quad \textsc{SWait} & i \triangleq i \leq 0 \\
\mathtt{af}_{\mathtt{QSem}} & \textsc{SSignal} & i & \triangleq (i+1, ()) & \mathtt{wa}_{\mathtt{QSem}} \quad \textsc{SSignal} & i \triangleq \textsc{False}
\end{array}$$

$$\begin{array}{llllll}
\mathtt{pf}_{\mathtt{QSem}} & \mathrm{L} & b & \triangleq (b, \textsc{False}) & \mathtt{ap}_{\mathtt{QSem}} \quad b & \textsc{SWait} \triangleq b \\
\mathtt{pf}_{\mathtt{QSem}} & \mathrm{R} & b & \triangleq (\textsc{False}, b) & \mathtt{ap}_{\mathtt{QSem}} \quad b & \textsc{SSignal} \triangleq \textsc{True}
\end{array}$$

Figure 6.4: $\mathfrak{qsem}$ – a quantity semaphore

Proving $c_l \sqsubseteq_{\mathfrak{qsem}} c$ and $c_r \sqsubseteq_{\mathfrak{qsem}} c$ is not hard. $c_r \sqsubseteq_{\mathfrak{qsem}} c$ holds because $c_r = c$, and $c_l \sqsubseteq_{\mathfrak{qsem}} c$ holds because context c_l permits only SSIGNAL, and context c, whatever it is, will also permit this. SWAIT cannot be performed concurrently with SWAIT, so when proving $c_l \Diamond_{\mathfrak{qsem}} c_r$ we need to prove $|||_{\mathfrak{qsem}}$ and $\mathsf{ally}_{\mathfrak{qsem}}$ for the remaining two valid pairs of actions.

- SSIGNAL and SSIGNAL. SSIGNAL $|||_{\mathfrak{qsem}}$ SSIGNAL holds because incrementing a counter is order independent. $\mathsf{ally}_{\mathfrak{qsem}}(\text{SSIGNAL}, \text{SSIGNAL})$ is trivially true because SSIGNAL is never stalled.

- SSIGNAL and SWAIT. SSIGNAL $|||_{\mathfrak{qsem}}$ SWAIT is true because the resultant effect in both cases is that the counter remains unchanged. $\mathsf{ally}_{\mathfrak{qsem}}(\text{SSIGNAL}, \text{SWAIT})$ holds because incrementing a counter cannot cause it to become 0 or negative if it was previously positive. $\mathsf{ally}_{\mathfrak{qsem}}(\text{SWAIT}, \text{SSIGNAL})$ holds because SSIGNAL is never stalled.

□

6.4 A unified I/O library

Finally, using the combinators defined in Chapter 5 we can combine all our small models into a single unified I/O model $\mathfrak{io}$ as follows:

$$\mathfrak{qsem}' \triangleq \mathtt{MtoLM}\ \mathfrak{qsem} \qquad \mathfrak{chan}' \triangleq \mathtt{MtoLM}\ \mathfrak{chan} \qquad \mathfrak{term}' \triangleq \mathtt{MtoLM}\ \mathfrak{term} \qquad ch_0 = ([], \text{CHOPEN})$$

$$\mathfrak{io}' \triangleq (\mathfrak{term}' *' \mathtt{smap}'\ \mathfrak{file}) *' (\mathtt{dmap}'\ ch_0\ \mathfrak{chan}' *' \mathtt{dmap}'\ 0\ \mathfrak{qsem}')$$

$$\mathfrak{io} \triangleq \mathtt{LMtoM}\ \mathfrak{io}'$$

This unified model lets the user

- do terminal I/O (via $\mathfrak{term}$, defined in Chapter 2),

- access a potentially infinite number of files, each identified by a different string,

- dynamically allocate communication channels, whose initial world state is ch_0, an empty open channel, and

- dynamically allocate quantity semaphores with initial value 0.

One can also, when splitting contexts, allocate to a process access to a particular file, a collection of files, or, perhaps, full access to one file, read access to another and write access to a particular channel.

The type of the model $\mathfrak{io}$' is enormous:

$$
\begin{aligned}
\mathfrak{io}' \;::\; & \mathsf{IOModelL} \\
& \mathsf{Either}\ (\mathsf{Either}\ \nu_{\mathtt{Term}}\ \nu_{\mathtt{File}})\ (\mathsf{Either}\ (\mathsf{Either}\ \nu_{\mathtt{Chan}}\ \mathsf{HndP})\ (\mathsf{Either}\ \nu_{\mathtt{QSem}}\ \mathsf{HndP})) \\
& \mathsf{Either}\ (\mathsf{Either}\ \alpha_{\mathtt{Term}}\ (\mathsf{String}, \alpha_{\mathtt{File}})) \\
& \quad (\mathsf{Either}\ (\mathsf{DynAction}\ \mathsf{HndP}\ \alpha_{\mathtt{Chan}})\ (\mathsf{DynAction}\ \mathsf{HndP}\ \alpha_{\mathtt{QSem}})) \\
& ((\rho_{\mathtt{Term}}, [(\mathsf{String}, \mathsf{Splitter}\ \rho_{\mathtt{File}})]), ([(\mathsf{HndP}, \mathsf{Splitter}\ \rho_{\mathtt{Chan}})], [(\mathsf{HndP}, \mathsf{Splitter}\ \rho_{\mathtt{QSem}})])) \\
& ((\omega_{\mathtt{Term}}, \mathsf{String} \to \omega_{\mathtt{File}}), (\mathsf{Pool}\ \omega_{\mathtt{Chan}}, \mathsf{Pool}\ \omega_{\mathtt{QSem}})) \\
& ((\varsigma_{\mathtt{Term}}, \mathsf{String} \to \mathsf{Cxt}\ \varsigma_{\mathtt{File}}), (\mathsf{HndP} \to \mathsf{Cxt}\ \varsigma_{\mathtt{Chan}}, \mathsf{HndP} \to \mathsf{Cxt}\ \varsigma_{\mathtt{QSem}}))
\end{aligned}
$$

This is cumbersome to use directly, but this can be solved by providing a set of libraries to give a more usable interface.

6.4.1 Semantics of Haskell's I/O actions

The actions in Figure 6.2 are implemented in model $\mathfrak{io}$ by first giving a definition to the Handle type.

$$
\begin{aligned}
\mathsf{Handle} \;\triangleq\; & \textsc{StdInHnd} \mid \textsc{StdOutHnd} \mid \textsc{FileHnd}\ \mathsf{String}\ (\mathsf{WPtr} \mid \textsc{RPtr}\ \mathsf{HndP}) \\
\mid\; & \textsc{ChanRdHnd}\ \mathsf{HndP} \mid \textsc{ChanWrHnd}\ \mathsf{HndP} \\
\mid\; & \textsc{QSemRdHnd}\ \mathsf{HndP} \mid \textsc{QSemWrHnd}\ \mathsf{HndP}
\end{aligned}
$$

A handle, in a nutshell, identifies something in the unified I/O model which may be written to or read from. With `stdin`, `stdout` and files this behaves as expected, but in the interface to $\mathfrak{io}$ a handle may also refer to a communication channel or a quantity semaphore.

Figures 6.5 and 6.6 give the implementations[‡] of a select number of actions from Figure 6.2 and two extra actions, `newChannel` and `hAllowed`. The remaining omitted definitions may be found in Appendix A.3.

What is important to note is that we can write whatever libraries or programs we want for the $\mathfrak{io}$ model and we do not have to worry about non-determinism. Similarly, one can bypass the libraries altogether. Confluence is now fully guaranteed in either case. The world state can still only ever be modified using a primitive action of the form `action` a. The definitions can also be seen as giving a full semantics to the various Haskell I/O actions, and this, I believe, may be one of the most useful contributions of this dissertation. Usually actions such as `openFile` are just defined as primitive, yet now we have given a rigorous semantics which actually attempts to explain its behaviour.

The command `openFile` n m attempts to open file n with I/O mode m. If m is

[‡]The functions `actionL` and `testL`, used in the implementations, are defined on page 90.

ReadMode it internally performs the two-step process for allocating a new read pointer and returns the new handle. Otherwise it attempts to open the file for writing, and this requires access to the whole file. The other file actions have roughly their desired effect as specified in the Haskell Report.

As mentioned earlier in the chapter, `newChannel` and `newQSem` dynamically create new communication channels or quantity semaphores, returning the respective read and write handle. The read and write handles associated with channels are different to file-handles in that they are not seekable, and the user may only close a write handle. If h is a read handle associated with a channel then `hIsEOF` h will return true if and only if the channel has been closed by the writer and there are no more characters to read from it. The interface to quantity semaphores is extremely limited. A `putChar` to a quantity semaphore signals it, ignoring the sent character, and a `getChar` performs a wait action, returning some constant, token character which the user will presumably just ignore. Since quantity semaphores can have multiple "writers" there is no easy way of implementing closing – how do we keep track to all the different writing processes and determine which have closed and which haven't? Handles which refer to quantity semaphores cannot, therefore, be closed.

6.4.2 Using handles to organise concurrency

We can now define `parIO`, a useful high-level interface to `par` for the `io` model. This allows one to control the resources given to each concurrent sub-program by simply listing the handles which each side requires access to. The full implementation details are rather involved and can be found in Appendix A.3. As per usual, the default behaviour is to give most permissions to the right-hand process. As mentioned previously, we don't want all permissions to go to one side by default since we want both processes to be able, say, to allocate any number of new communication channels. Also, unlike `par` which requires an extra function as parameter, `parIO` returns a tuple containing the return values of the left and right process. This, in general, is more convenient.

```
parIO :: ∀β. ∀γ. [Handle] → Prog_io β → [Handle] → Prog_io γ → Prog_io (β,γ)
parIO hsl ml hsr mr = par ((tp,fsp),(chp,qsp)) ml mr (\vl vr -> (vl,vr))
 where
   tp  = splitHndsTerm hsl hsr
   fsp = splitHndsFiles hsl hsr
   chp = splitHndsChans hsl hsr
   qsp = splitHndsQSems hsl hsr
```

Sometimes we want to give a sub-process access to an entire file n. This can be achieved using the "handle" `wholeFile` n. A process must have complete access to a file before it may be allowed to write to it, and `wholeFile` n is `FileHnd n WPtr`, the single write pointer for that file.

```
stdin, stdout :: Handle
stdin  = StdInHnd
stdout = StdOutHnd

openFile :: FilePath → IOMode → Prog_io Handle
openFile n ReadMode = do
  Left (Right (RHnd h)) <- actionL (Left (Right (n,FNextRdPtr)))
  actionL (Left (Right (n, FRdOpen h)))
  return (FileHnd n (RPtr h))
openFile n m = do
  actionL (Left (Right (n, WholeFileAct (HWrOpen m))))
  return (FileHnd n WPtr)

hGetChar :: Handle → Prog_io Char
hGetChar StdInHnd = do
  Left (Left c) <- actionL (Left (Left GetC))
  return c
hGetChar (FileHnd n p) = do
  Left (Right (RChar c)) <-
    actionL (Left (Right (n,FilePtrAct p FGetChar)))
  return c
hGetChar (ChanRdHnd h)   =
  actionL (Right (Left (DynAct h ChRcve))) >>=
    \(Right (Left (Left (Just c)))) -> return c
hGetChar (QSemRdHnd h) =
  actionL (Right (Right (DynAct h SWait))) >> return 'X'

hPutChar :: Handle → Char → Prog_io ()
hPutChar h c = do
  case h of
    StdOutHnd        -> actionL (Left (Left (PutC c)))
    (FileHnd n WPtr) ->
      actionL (Left (Right (n,WholeFileAct (FPutChar c))))
    (ChanWrHnd h)    -> actionL (Right (Left (DynAct h (ChSend c))))
    (QSemWrHnd h)    -> actionL (Right (Right (DynAct h SSignal)))
  return ()

hClose :: Handle → Prog_io ()
hClose (FileHnd n p) = do
  actionL (Left (Right (n, FilePtrAct p FClosePtr)))
  return ()
hClose (ChanWrHnd h)   = do
  actionL (Right (Left (DynAct h ChClose)))
  return ()
hClose _             = return ()
```

Figure 6.5: Implementation of unified I/O libraries (Part I)

```
removeFile :: FilePath → Prog_io ()
removeFile n = do
  actionL (Left (Right (n,WholeFileAct FRemove)))
  return ()

hIsEOF :: Handle → Prog_io Bool
hIsEOF (FileHnd h p) = do
  Left (Right (RBool b)) <- actionL (Left (Right (n,FilePtrAct p FIsEOF)))
  return b
hIsEOF (ChanRdHnd h)   = do
  Right (Left (Left mc)) <- actionL (Right (Left (DynAct h ChLook)))
  return (isNothing mc)
hIsEOF _              = return False

hIsOpen :: Handle → Prog_io Bool
hIsOpen (FileHnd n p) = do
  Left (Right (RBool b)) <- case p of
    RPtr h -> actionL (Left (Right (n, FRdOpen h)))
    WPtr   -> actionL (Left (Right (n, WholeFileAct HWrOpen)))
  return b
hIsOpen (ChanWrHnd h)        = do
  Left (Right (Left c)) <- actionL (Right (Left (DynAct h ChIsClosed)))
  return (c=='O')
hIsOpen _                    = return True

newChannel :: Prog_io (Handle,Handle)
newChannel = do
  Right (Left (Right h)) <- actionL (Right (Left Next))
  actionL (Right (Left (Alloc h)))
  return (ChanRdHnd h, ChanWrHnd h)

hIsWritable :: Handle → Prog_io Bool
hIsWritable h = case h of
  StdInHnd           -> return False
  ChanRdHnd h        -> return False
  QSemRdHnd h        -> return False
  FileHnd n (RPtr h) -> return False
  _                  -> return True

hAllowed :: Handle → Prog_io Bool
hAllowed StdInHnd         = testL (Left (Left GetC))
hAllowed StdOutHnd        = testL (Left (Left (PutC 'X')))
hAllowed (FileHnd n WPtr) = testL (Left (Right (n, FPutChar 'X')))
hAllowed (FileHnd n (RPtr h)) =
                     testL (Left (Right (n, FilePtrAct (RPtr h) FGetChar)))
hAllowed (ChanWrHnd h)    = testL (Right (Left (DynAct h (ChSend 'X'))))
hAllowed (ChanRdHnd h)    = testL (Right (Left (DynAct h ChRcve)))
hAllowed (QSemWrHnd h)    = return True
hAllowed (QSemRdHnd h)    = testL (Right (Right (DynAct h SWait)))
```

Figure 6.6: Implementation of unified I/O libraries (Part II)

```
wholeFile :: FilePath → Handle
wholeFile n = FileHnd n WPtr
```

Occasionally, also, we will want to allow both sub-processes to be able to open a file for reading even though we do not, as yet, possess a read handle for that file. One can solve this with a dummy read handle `readAccess` n. The HndP of value $([], -1)$ cannot refer to an existing read pointer, and any attempt to use it will fail outright. However it is still possible that a context can *permit* it. By using this handle we can avoid the default behaviour whereby if the parent process has complete access to the file then the left-hand process receives no permissions at all.

```
readAccess :: FilePath → Handle
readAccess n = FileHnd n (RPtr ([],-1))
```

6.4.3 Other issues

We include library functions to check if the I/O context permits access to certain handles or files but we omit any library calls which check if the *allocation* of a new channel or quantity semaphore is legal. It is assumed that under normal circumstances this will not be necessary. In other words, we presume that a program will only use a handle returned by an existing library call such as `newChannel`, and not write something arbitrary such as:

```
parIO [ChanRdHnd ([L,R,L],22)] p1 [QSemWrHnd ([R],345)] p2
```

which may cause a process not to have access to the very handle that a call to `newChannel` would return.

For this to work we need an intuition concerning the outermost I/O context in which a program runs. In general it can be assumed that each program begins with access to all resources – every channel or semaphore handle, both terminal devices, and complete access to all files. With regard to files this could possibly be weakened. Perhaps we would want to exclude files which, at runtime, the program does not have access to.

6.5 Applications

At long last we can now give a collection of real applications for Curio. All the applications, as one might expect, make use of the fact that over-specifying the sequence in which actions are performed can be inconvenient and inelegant.

6.5.1 A file encoder with user interface

For the first and most comprehensive example, let us say that I want to write a file encoder which has an accompanying user interface. The encoder and the interface should probably be understood as two relatively distinct pieces of code – one reads and writes to files, and the other communicates with the user. Yet if actions are sequenced explicitly as they are

in plain Haskell then either user input will have to halt while a file is being encoded, or the two applications will have to be merged together somehow. Concurrent Haskell, however, would be overkill – we know that our program should be deterministic and we want the language and runtime system to enforce this. Lazy file I/O such as `hGetContents` would be inadvisable since the files in question could be enormous.

Curio may be used to solve this problem. The skeleton of this kind of file encoder can be found in Figures 6.7 and 6.8[§]. We omit all details of the actual encoding algorithm because this is not the point of the exercise – what is important is that encoding a file requires access to one or two files and may take a long time. `encoderMain` starts the program and is mainly a wrapper for `encoderUI`. `encoderUI` continually reads input commands from the user and performs them whilst maintaining an (initially empty) list of the processes running concurrently. The commands are:

- Encode a source file to a target file, with some options. The I/O context must permit read access to the source file and write access to the target file. First a channel is created to allow the encoding process to communicate with the user interface. Execution is then split using `parIO`.

  The left-hand process runs `encodeFile` and is explicitly given read access to the source file, write access to the target file, and the right to send data along the new channel. Each time a block of data is encoded it sends a '`.`' along the channel and closes the channel when finished.

  The right-hand process gets the permissions that are left over, as is always the case with `parIO`. It may do one of two things depending what the user specified:

  - Sequential encoding. The right-hand process merely waits until the file is encoded, printing the '`.`'s sent from the encoding process. This is achieved using `joinHnds` which connects the encoding process's channel to `stdout` until the channel is closed.
  - Concurrent encoding. The right-hand process immediately begins accepting user input again, adding the concurrent process's details to the list that it maintains.

- Wait for a target file to be encoded. This reads the characters from that file's associated communication channel, terminating when the channel is closed.

- List the processes which are running.

- Exit. If this command is entered then once all concurrent processes have terminated the program execution will return finally to `encoderMain`, which was the original caller of `encoderUI`.

[§]The implementation makes use of the standard Haskell functions `putStr` and `putStrLn`. These write a string to `stdout` with or without a carriage return respectively, and both are defined using `hPutChar`.

```
encoderMain :: Prog_io ()
encoderMain = do
  b1 <- hAllowed stdin
  b2 <- hAllowed stdout
  if (not (b1 && b2)) then (return ()) else (encoderUI [])

encoderUI :: [(FilePath,FilePath,Handle)] → Prog_io ()
encoderUI hnds = do
  putStr "Enter next command: "        -- print user prompt
  cmd <- readEncoderCmd                -- get command from user
  case cmd of
     -- encode file 'sf' to file 'tf' with various options.
    Encode sf tf mode encoding_options -> do
      b1 <- fAllowedR sf               -- do we have read access to sf?
      b2 <- fAllowedW tf               -- .. write-access to tf?
      b3 <- doesFileExist sf           -- and does sf exist?
      if (not (b1 && b2 && b3))
        then (do
          putStrLn stdout "Access denied."
          encoderUI hnds)              -- try again if not
        else (do
          (irh,iwh) <- newChannel      -- otherwise, make a channel
          parIO                        -- initiate concurrency
            [wholeFile tf, readAccess sf, iwh]
            (encodeFile sf tf iwh encoding_options) -- LH: encode the file
            [irh,stdin,stdout]
            (case mode of              -- the RH process:
               Sequential -> do        -- if we want seqential encoding..
                 putStr ("Encoding "++sf)
                 joinHnds irh stdout   -- connect the handle irh to stdout
                 putStrLn " Done!"     -- (show '.' for each encoded block)
                 encoderUI hnds        -- when finished resume UI
               Concurrent ->
                 -- for concurrent encoding resume interaction immediately
                 -- and add to 'hnds'
                 encoderUI ((sf,tf,irh):hnds)))
     -- wait until file 'tf1' has been fully written to
    WaitFor tf1   -> do
      putStr ("Waiting for "++sf1)
      joinHnds (hndsTFLkp hnds tf1) stdout -- for each block written to tf1
      putStrLn " Done!"                    --   print a '.'
      encoderUI hnds                       -- ... then resume user input
     -- display all elements of 'hnds' to the user
    ListProcesses -> putStr (prettifyHandleList hnds) >> encoderUI hnds
     -- exit encoder. This will propagate back to encoderMain once all
     -- concurrent processes are complete.
    ExitEncoder   -> return ()
```

Figure 6.7: A file encoder in Curio (Part I)

```
encodeFile :: FilePath → FilePath → Handle → EncoderOptions → Prog_io ()
encodeFile srcfile tgtfile infoch opts = do
    removeFile tgtfile               -- delete the target file
    ht <- openFile tgtfile WriteMode -- create target for writing
    hs <- openFile srcfile ReadMode  -- open source for reading

    -- prepare the target for encoding (no implementation given)
    encoderLoop ht hs infoch opts  -- perform encoding
    -- finalise (no implementation given)

    hClose infoch               -- close the communication channel
    hClose hs                   -- close the source read handle
    hClose ht                   -- close the target write handle
    return ()
  where
    encoderLoop ht hs hi = do
      b <- hIsEOF hs                -- if EOF then stop
      if b
        then (return ())
        else (do
          -- read a block from 'hs', encode, and write to 'ht'
          -- (no implementation given)
          hPutChar hi '.'           -- signal to UI with a '.'
          encoderLoop ht hs hi)     -- .. then repeat

joinHnds :: Handle → Handle → Prog_io ()
-- 'connect' a read handle directly to a write handle. It returns once the
-- read handle has been closed, but does not close the write handle
joinHnds rh wh f = do
  b <- hIsEOF rh         -- if there is no more input
  if b                   -- on the read handle
    then (return ())     -- ... then finish
    else (do
      c <- hGetChar rh   -- otherwise read 'c'
      hPutChar wh c      -- write 'c'
      joinHnds rh wh f)  -- .. and repeat


          EncoderCmd  ::  Encode FilePath FilePath EncoderMode EncoderOptions
                      |   WaitFor FilePath | ListProcesses | ExitEncoder
         EncoderMode  ::  Sequential | Concurrent
      readEncoderCmd  ::  Prog_io EncoderCmd
  prettifyHandleList  ::  [(FilePath,FilePath,Handle)] → String
```

Figure 6.8: A file encoder in Curio (Part II)

There are some important points worth noting:

- It is possible for a single file to be encoded concurrently by many processes (presumably with different options) just as long as the target files differ in each occurrence. Without shared reading this would not be possible.
- `encoderUI` can only terminate once all created encoder processes have also terminated. Therefore, once a file is opened for writing then one will not be able to reopen it until `encoderUI` has terminated.
- We check the I/O context to guarantee that we have access to the various files (which internally makes use of the `test` primitive) and we also check if the source file actually exists (which is, internally, a primitive `action` which returns information about the world state). This demonstrates once again that I/O contexts are entirely orthogonal to the dynamic properties of a file system.

6.5.2 A background log file

Imagine that one wishes to write a large, I/O-intensive application which also maintains a log file of the various actions it performs. This may be achieved sequentially by running a particular monadic I/O program `logData` each time new data is ready to be logged.

```
logData :: LogStructure -> IO ()
```

However, there are a number of disadvantages to this:

- The `logData` function could be non-trivial and need to maintain local state which must be accessed and modified every time the function is called. For example, each time some data is to be logged it may be added to a complex internal structure before, after some time, being written to disk. If this is the case then `logData` will have to maintain this structure in global state somehow (for example, using Haskell's `IORef`s).
- Logging data may possibly be time-consuming and we do not want to force the rest of the program to freeze when logging is in progress. We may instead require the logging of data to take a lower precedent compared to that of the rest of the application.
- If the main application is suspended for long periods of time, perhaps waiting for user input, we may want any buffered log data still be processed regardless after a specific period of idle time.

On the whole this sequential approach is a poor design strategy. What we want is to run the logging process concurrently with the rest of the application and have the language guarantee that program execution will be deterministic. In Curio this may be achieved quite simply as follows:

```
logApplicationMain :: Prog_io ()
logApplicationMain = do
  (logr, logw) <- newChannel
  parIO
    [wholeFile "logFile.txt", logr] (logProcess logr)
    [logw]                          (applProcess logw)
  return ()
```

`logProcess` repeatedly reads information from handle `logr`, processes it and writes it to file "`logFile.txt`". The rest of the application is contained in `applProcess` and this behaves as normal with the exception that any data for logging will be written to handle `logw`. Once `logApplicationMain` terminates both concurrent processes will be guaranteed to have also finished.

We do not mention how the logging process might lower its priority or pause for a number of milliseconds. Concurrent Haskell has facilities like this, and introduces some extra primitives such as `yield`, which forces a context switch to occur, and `threadDelay`, which delays a process for a specific number of milliseconds. Our semantics is too high-level to even begin to describe this type of behaviour usefully. With confluence, however, we can be certain that these sort of actions – those actions that merely change the order in which concurrent processes are interleaved – would be semantically transparent in a real implementation.

6.5.3 Concurrent file processing

An advantage to not over-sequencing actions is that one may process individual files in a concurrent fashion. Not only is this a clearer specification of the program's desired behaviour, but by loosening the order in which actions must be performed a clever implementation may be able to optimise the program's performance.

`foldlFile`, defined in Figure 6.9, is a general function for scanning a file. The program `foldlFile` f b_0 n performs a `foldl` operation on the files contents, but behaves sensibly, returning b_0, if the file is locked out by the sub-program's I/O context or it doesn't exist. We can use `foldlFile` to write programs `wordCount` and `getFContents`, shown below, which, respectively, counts the number of words in a file and (strictly) reads the entire file contents into a list.

```
wordCount :: FilePath → Prog_io Int
wordCount n = do
    (count,inword) <- foldlFile wordCountTail (0,False) n
    return count
  where
    wordCountTail (count,inword) c =
      (count + if (inword && isSpace c) then 1 else 0, isSpace c)
```

```
foldlFile :: ∀β. (β → Char → β) → β → FilePath → Prog_io β
foldlFile f b0 n = do
    canread <- fAllowedR n                  -- can I read from 'n'?
    exists  <- doesFileExist n              -- and does it exist?
    if (not (canread && exists))
      then (return b0)
      else (do h <- openFile n ReadMode     -- if so, open it
               b <- foldlHandle f b0 h      -- process the file
               hClose h                     -- close it
               return b)                    -- then return the result
  where
    foldlHandle f b0 h = do
      eof <- hIsEOF h
      if eof                                -- if EOF
        then (return b0)                    --   return b0
        else (do c <- hGetChar h            --   otherwise get a Char
                 foldlHandle f (f b0 c) h)  --   and continue, updating b0
```

Figure 6.9: Definition of `foldlFile`

```
getFContents :: FilePath → Prog_io [Char]
getFContents n = foldlFile (flip (:)) [] n
```

All the previous functions could have just as easily been written in Haskell, but now we can make use of deterministic concurrency. `mapFile` takes a function f and a list of file paths ns as an argument and concurrently, for each file n in ns, runs the I/O performing program $f\ n$ attempting to give it complete access to file n. In the event that a particular name n appears twice in the list then the second of the two processes will have no access to file n.

```
mapFile :: ∀β. (FilePath → Prog_io β) → [FilePath] → Prog_io [β]
mapFile f []     = return []
mapFile f (n:ns) = do
  (b,bs) <- parIO [wholeFile n] (f n) [] (mapFile f ns)
  return (b:bs)
```

So, for example, `wordCounter` concurrently counts the number of words in the files associated with a list of filenames.

```
wordCounter :: [FilePath] → Prog_io [Int]
wordCounter ns = mapFile wordCount ns
```

6.5.4 Quantity semaphore example

As admitted previously, quantity semaphores are a somewhat weak communication primitive because the one reading process is not able to distinguish the many writing processes. They do have their uses, however. If multiple files are being processed concurrently then the user

may want visual feedback concerning the number of files that have yet to be fully scanned. A quantity semaphore is an ideal tool for this, since we are not required to ask *which* files have been fully read (this could be different on each program execution).

`waitFor` i h waits for i characters on handle h and then exits, and each time a character is read it prints to `stdout` the number remaining.

```
waitFor :: Int → Handle → Prog_io ()
waitFor i h | i<=0      = putStrLn "Finished!" >> return ()
waitFor i h | otherwise = do hGetChar h
                             putStrLn (show i ++ " yet to do...")
                             waitFor (i-1) h
```

`wordCountStdout` ns concurrently performs a word count on a list of files and also indicates to the user on `stdout` each time a file has been scanned, eventually printing all the respective word counts, and a sum total.

```
wordCountStdout :: [FilePath] → Prog_io ()
wordCountStdout ns = do
  (qsr,qsw) <- newQSem
  (_,is)    <- parIO
    [qsr, stdout] (waitFor (length ns) qsr)
    [qsw]         (mapFileWith [qsw] (\n -> do
                                        i <- wordCount n
                                        hPutChar qsw 'X'
                                        return i)
                               ns)
  putStrLn ("Individual word counts: " ++ show (zip ns is))
  putStrLn ("Total words: " ++ show (sum is))
  return ()
```

`mapFileWith` is a small, easy modification to the `mapFile` function shown earlier. It takes an extra argument hs, a list of handles, and it attempts to give each created subprocess access to these handles. Even though a quantity semaphore may have an unlimited number of writers, the default, when performing concurrency, is still just to blindly give one process exclusive access to the structure.

6.6 Chapter summary

This chapter presented a full I/O model $\mathfrak{io}$ which gave the semantics to a subset of Haskell's API extended with deterministic concurrency communication primitives. The chief component I/O models were $\mathfrak{file}$, $\mathfrak{term}$, $\mathfrak{chan}$ and $\mathfrak{qsem}$, and these were combined using the location-based combinators described in Chapter 5. The chapter concluded with four example applications which harnessed the power of the new, extended API.

Having shown that Curio's I/O models can handle substantial, real world I/O, we now turn our back on these specific capabilities of Curio. The final two technical chapters of this dissertation are concerned with semantics properties of the general Curio language and I/O model structure. The following chapter, Chapter 7, gives a semantic model which describes the behaviour of arbitrary Curio programs.

Chapter 7

Axiomatic semantics

Having proved confluence in Chapter 4 we are now in the powerful position of being able to identify a world/program pair solely by its resultant normal form, if one exists. In this chapter we build upon this theory to attempt to give a full axiomatic semantics for Curio.

An axiomatic semantics [43, 124] can be understood as the act of describing a language by stating the properties which hold for terms of the language. Unlike an operational or denotational semantics, an axiomatic semantics is somewhat high-level and vague since it doesn't attempt to actually explain the behaviour of terms. In some sense, however, the ultimate goal of language design is a rich axiomatic semantics, since if programs cannot be manipulated and understood at a high-level then the language is most likely neither modular or compositional.

We begin in Section 7.1 by constructing a big-step operational semantics for Curio, allowing one to relate the evaluation of a term's sub-terms to the evaluation of that term as a whole. Section 7.2 uses this to develop a co-inductive definition of what it means for two programs to behave the same with respect to some given I/O context. This definition defines the meaning of a program fragment in terms of both how it affects world state, and how it responds to arbitrary changes to world state made by other processes. In Section 7.3 we generalise this relation to all I/O contexts and use it to prove some important results, such as the monad laws. Also demonstrated is the fact that the equivalence relation is a congruence with respect to the I/O constructs in the language. This effectively shows that it is a *useful* equality relation, since in this way equivalent programs may be substituted for one another within a larger program. Section 7.4 gives some equivalence proofs for programs which perform actions, one of which is non-trivial. Section 7.5 concludes the chapter with a discussion about real world proofs and how the language design affects the ease with which properties are proved.

Due to the denotational style which we adopt when giving the operational semantics to Curio, we are unable to machine-verify some higher-order properties. These are shown to be true "by hand".

Note: For all of this chapter we assume implicitly that $\mathsf{PRE}_{\mathsf{s}}$ holds.

7.1 A big-step semantics

A big-step semantics is used to demonstrate that the normal form of some term t in a language can be somehow understood with respect to the normal forms of t's sub-terms. Although a single-step semantics gives an excellent intuition for how a program executes over time, it is somewhat low-level. By proving that world/program pairs can be identified by their normal forms, or lack thereof, we have effectively shown that individual single-step reduction does not really matter, which in turn suggests that the language semantics should be re-written using $\Downarrow^c$ and $\Uparrow^c$ rather than $\longrightarrow^c$, $\downarrow^c$ and $\uparrow^c$. This is the goal of this section.

7.1.1 Preliminaries

We begin by proving some initial results, mainly concerning `>>=` and `par`.

Having proved confluence, the operator $\Downarrow$ can now safely replace any occurrence of $\twoheaddownarrow$. At times, however, we need to retain $\twoheadrightarrow$, and this is still non-deterministic. Even with confluence, $w \Vdash m \twoheadrightarrow^c w_1 \Vdash m_1$ only indicates that, in the process of reducing to some normal form $w_2 \Vdash m_2$ or diverging, that it is possible that one can "pass through" world/program pair $w_1 \Vdash m_1$.

Proposition 7.1.1. *$w \Vdash m \overset{i_1+i_2}{\Downarrow^c} w_2 \Vdash m_2$ if and only if there is some (not necessarily unique) $w_1 \Vdash m_1$ such that $w \Vdash m \overset{i_1}{\twoheadrightarrow^c} w_1 \Vdash m_1$ and $w_1 \Vdash m_1 \overset{i_2}{\Downarrow^c} w_2 \Vdash m_2$.*

Proof. Both sides of the implication can be proved by showing that, assuming confluence, $\Downarrow^c$ is always completely equivalent to $\twoheaddownarrow^c$. Therefore, expanding the definition of $\twoheaddownarrow^c$, it becomes a proof of the following: $w \Vdash m \overset{i_1+i_2}{\twoheadrightarrow^c} w_2 \Vdash m_2$ and $w_2 \Vdash m_2 \downarrow^c$ if and only if there is some (not necessarily unique) $w_1 \Vdash m_1$ such that $w \Vdash m \overset{i_1}{\twoheadrightarrow^c} w_1 \Vdash m_1$ and $w_1 \Vdash m_1 \overset{i_2}{\twoheadrightarrow^c} w_2 \Vdash m_2$ and $w_2 \Vdash m_2 \downarrow^c$. This is obviously true from the transitivity of $\twoheadrightarrow^c$. □

The above proposition, used quite often below, is a straightforward property of confluence: if a program always converges to the same normal form, then if we fix the first i_1 reduction steps then it will still, also, always converge to that same normal form. This is natural since removing some non-determinism cannot make reduction more unpredictable.

Lemma 7.1.1. *If $w \Vdash m_l \overset{i}{\Downarrow^{c_l}} w' \Vdash m'_l$, and assuming `pf p c = (`$c_l$`,`$c_r$`)` and $m_r \neq \bot$, then $w \Vdash m_l \mid\mid\mid^p_* m_r \overset{i}{\twoheadrightarrow^c} w' \Vdash m'_l \mid\mid\mid^p_* m_r$.*

Proof. Induction on i. Base case ($i = 0$): trivial, since $w' = w$, $m'_l = m_l$ and $\twoheadrightarrow$ is reflexive. Inductive case ($i = k + 1$): it is true that $w \Vdash m_l \longrightarrow^{c_l} w'' \Vdash m''_l$ and $w'' \Vdash m''_l \overset{k}{\Downarrow^{c_l}} w' \Vdash m'_l$ for some (not necessarily unique) w'' and m''_l (Proposition 7.1.1). Apply IH to prove $w'' \Vdash m''_l \mid\mid\mid^p_* m_r \overset{k}{\twoheadrightarrow^c} w' \Vdash m'_l \mid\mid\mid^p_* m_r$, and from the language semantics $w \Vdash m_l \mid\mid\mid^p_* m_r \longrightarrow^c w'' \Vdash m''_l \mid\mid\mid^p_* m_r$. From the transitivity of $\twoheadrightarrow$, prove $w \Vdash m_l \mid\mid\mid^p_* m_r \overset{k+1}{\twoheadrightarrow^c} w' \Vdash m'_l \mid\mid\mid^p_* m_r$. □

Lemma 7.1.2. *If* $w \Vdash m_l \ \texttt{|||}^p_* \ m_r \ {}^i\Downarrow^c \ w'' \Vdash m''$ *and* $\texttt{pf}\ p\ c = (c_l, c_r)$ *then there is some* $w' \Vdash m'_l$ *and* i_1 *such that* $w \Vdash m_l \ {}^{i_1}\Downarrow^{c_l} \ w' \Vdash m'_l$.

Proof. Induction on i. Base case ($i = 0$): $w'' = w$ and $m'' = m_l \ \texttt{|||}^p_* \ m_r$ and since it is in normal form, m_l is also in normal form in context c_l. Let $i_1 = 0$, $w = w'$ and $m_l = m'_l$. Inductive case ($i = k + 1$): if $w \Vdash m_l$ is in normal form in context c_l then let $i_1 = 0$ like in the base case. Otherwise $w \Vdash m_l \longrightarrow^{c_l} w''' \Vdash m'''_l$ for some w''', m'''_l. From the language semantics this implies $w \Vdash m_l \ \texttt{|||}^p_* \ m_r \longrightarrow^c w''' \Vdash m'''_l \ \texttt{|||}^p_* \ m_r$, and, from confluence, $w''' \Vdash m'''_l \ \texttt{|||}^p_* \ m_r \ {}^k\Downarrow^c \ w'' \Vdash m''$. Apply IH to prove that for some i'_1, w' and m'_l, $w''' \Vdash m'''_l \ {}^{i'_1}\Downarrow^{c_l} \ w' \Vdash m'_l$, and use confluence to prove that $w \Vdash m_l \ {}^{i'_1+1}\Downarrow^{c_l} \ w' \Vdash m'_l$ (that is, let $i_1 = i'_1 + 1$). □

Lemma 7.1.3. *If* $w \Vdash m_r \ {}^i\Downarrow^{c_r} \ w' \Vdash m'_r$, *and assuming* $\texttt{pf}\ p\ c = (c_l, c_r)$ *and* $m_l \neq \bot$ *then* $w \Vdash m_l \ \texttt{|||}^p_* \ m_r \ {}^i\twoheadrightarrow^c w' \Vdash m'_r \ \texttt{|||}^p_* \ m_r$.

Proof. Similar to proof of Lemma 7.1.1. □

Lemma 7.1.4. *If* $w \Vdash m_l \ \texttt{|||}^p_* \ m_r \ {}^i\Downarrow^c \ w'' \Vdash m''$ *and* $\texttt{pf}\ p\ c = (c_l, c_r)$ *then there is some* $w' \Vdash m'_r$ *such that* $w \Vdash m_r \Downarrow^{c_r} \ w' \Vdash m'_r$.

Proof. Similar to proof of Lemma 7.1.2. □

Lemma 7.1.5. *If* $w \Vdash m \ \texttt{>>=}\ f \ {}^i\Downarrow^c \ w'' \Vdash m''$ *then there is some* $w' \Vdash m'$ *and* i_1 *such that* $w \Vdash m \ {}^{i_1}\Downarrow^c \ w' \Vdash m'$ *and either*

- *There is a* v *such that* $m' = \texttt{return}\ v$ *and* $w' \Vdash f\ v \ {}^{i-i_1-1}\Downarrow^c \ w'' \Vdash m''$.
- m' *is not a value,* $w' = w''$, $m'' = m' \ \texttt{>>=}\ f$ *and* $i_1 = i$.

Proof. Induction on i.

- Base case ($i = 0$): this means $w'' = w$, $m'' = m \ \texttt{>>=}\ f$ and $w \Vdash m \ \texttt{>>=}\ f \downarrow^c$, which implies $w \Vdash m \downarrow^c$, since m is not a value. Therefore $w' = w$, $m' = m$ and $i_1 = 0$, and it is the second of the two possibilities.
- Inductive case ($i = k + 1$): we know $w \Vdash m \ \texttt{>>=}\ f \longrightarrow^c w''' \Vdash m'''$ and $w''' \Vdash m''' \ {}^k\Downarrow^c \ w' \Vdash m'$ for some $w''' \Vdash m'''$.
  - If m is some value $\texttt{return}\ v$ then $w''' = w$, $m''' = f\ v$ and $i_1 = 0$ (any value is in normal form after 0 steps, so $w' = w$ and $m' = m$), and it is the first possibility.
  - If m is not a value, then the reduced redex must be within m itself: $w \Vdash m \longrightarrow^c w''' \Vdash m'''_1$ and $m''' = m'''_1 \ \texttt{>>=}\ f$. In this case, apply IH to prove that for some $w' \Vdash m'$, i'_1 such that $w''' \Vdash m'''_1 \ {}^{i'_1}\Downarrow^c \ w' \Vdash m'$, and therefore $w \Vdash m \ {}^{i'_1+1}\Downarrow^c \ w' \Vdash m'$. Let $i_1 = i'_1 + 1$. Both of the two possibilities carry over directly from the inductive hypothesis.

□

Lemma 7.1.6. *If* $w \Vdash m \ ^{i}\Downarrow^{c} \ w' \Vdash m'$ *and* m' *is not a value then* $w \Vdash m \texttt{>>=} f \ ^{i}\Downarrow^{c} w' \Vdash m' \texttt{>>=} f$.

Proof. Induction on i. Base case ($i = 0$): $w = w'$, $m = m'$ and $w \Vdash m \downarrow^{c}$, and since m is not a value, $w \Vdash m \texttt{>>=} f \downarrow^{c}$. Inductive case ($i = k + 1$): $w \Vdash m \longrightarrow^{c} w'' \Vdash m''$ and $w'' \Vdash m'' \ ^{k}\Downarrow^{c} w' \Vdash m'$ for some w'', m'' (Proposition 7.1.1). Using IH, prove that $w'' \Vdash m'' \texttt{>>=} f \ ^{k}\Downarrow^{c} w' \Vdash m' \texttt{>>=} f$, and from the language semantics show $w \Vdash m \texttt{>>=} f \longrightarrow^{c} w'' \Vdash m'' \texttt{>>=} f$. Therefore, $w \Vdash m \texttt{>>=} f \ ^{k+1}\Downarrow^{c} \ w' \Vdash m' \texttt{>>=} f$. □

Lemma 7.1.7. *If* $w \Vdash m \ ^{i_1}\Downarrow^{c} \ w' \Vdash \texttt{return}\, v$ *and* $w' \Vdash f \ v \ ^{i_2}\Downarrow^{c} \ w'' \Vdash m''$ *then it is true that* $w \Vdash m \texttt{>>=} f \ ^{i_1+i_2+1}\Downarrow^{c} \ w'' \Vdash m''$.

Proof. Induction on i_1. Base case ($i_1 = 0$): $w' = w$, $m = \texttt{return}\ v$, and since we know that $w \Vdash \texttt{return}\ v \texttt{>>=} f \longrightarrow^{c} w \Vdash f\ v$, we can prove $w \Vdash \texttt{return}\ v \texttt{>>=} f \ ^{i_2+1}\Downarrow^{c} \ w'' \Vdash m''$. Inductive case ($i_1 = k_1 + 1$): for some $w''' \Vdash m'''$, $w \Vdash m \longrightarrow^{c} w''' \Vdash m'''$ and $w''' \Vdash m''' \ ^{k_1}\Downarrow^{c} w' \Vdash \texttt{return}\ v$. Apply IH to prove $w''' \Vdash m''' \texttt{>>=} f \ ^{k_1+i_2+1}\Downarrow^{c} \ w'' \Vdash m''$, and since from the semantics of `>>=`, $w \Vdash m \texttt{>>=} f \longrightarrow^{c} w''' \Vdash m''' \texttt{>>=} f$, it is true that $w \Vdash m \texttt{>>=} f \ ^{i_1+i_2+1}\Downarrow^{c} w'' \Vdash m''$. □

7.1.2 World/program equivalence

In order to proceed we develop an equivalence relation on world/program pairs. As a result of confluence it is natural to identify two world/program pairs in some context if either

1. they both converge to the same resultant world/program pair, or
2. they both diverge.

The $\simeq^{c}$ operator denotes this equivalence of two world/program pairs in context c, and its formal definition can be found in Figure 7.1. Despite this operator's existence as a half-way house, the seeds of a big-step semantics are to be found in the properties which it enjoys. If $w \Vdash m \ \Downarrow^{c} \ w_1 \Vdash m_1$ then it is obviously true that $w \Vdash m \ \simeq^{c} \ w_1 \Vdash m_1$. Therefore, by showing how the equivalence of two world/program pairs is preserved in the presence of `>>=`, `par`, and `test` we immediately know how program *reduction* is preserved in the presence of the various constructors – a program is equivalent to the normal form to which it reduces.

Proposition 7.1.2. $\simeq^{c}$ *is an equivalence relation (symmetric, reflexive and transitive).*

Proof. Clear from the definition. □

Proposition 7.1.3. *If* $w \Vdash m \twoheadrightarrow^{c} w_1 \Vdash m_1$ *then* $w \Vdash m \ \simeq^{c} \ w_1 \Vdash m_1$.

Proof. $w \Vdash m$ either converges or diverges.

$$w_1 \Vdash m_1 \ \simeq^c \ w_2 \Vdash m_2 \ \triangleq \ \begin{array}{c} w_1 \Vdash m_1 \Uparrow^c \ \wedge \ w_2 \Vdash m_2 \Uparrow^c \\ \vee \\ \exists_{w_3 \in \omega}.\exists_{m_3 \in \mathsf{Prog}\ \nu\ \alpha\ \rho.} \left(\begin{array}{c} w_1 \Vdash m_1 \Downarrow^c \ w_3 \Vdash m_3 \\ \wedge \\ w_2 \Vdash m_2 \Downarrow^c \ w_3 \Vdash m_3 \end{array} \right) \end{array}$$

Figure 7.1: Definition of world/program equivalence

$$\frac{w_1 \Vdash m_1 \ \simeq^c \ w_2 \Vdash m_2}{w_1 \Vdash m_1 \ \texttt{>>=}\ f \ \simeq^c \ w_2 \Vdash m_2 \ \texttt{>>=}\ f}$$

$$\texttt{pf}\ p\ c = (c_l, c_r) \quad \left\{ \begin{array}{l} \dfrac{w_1 \Vdash m_{l1} \ \simeq^{c_l} \ w_2 \Vdash m_{l2}}{w_1 \Vdash m_{l1} \ |||^p_* \ m_r \ \simeq^c \ w_2 \Vdash m_{l2} \ |||^p_* \ m_r} \\ \\ \dfrac{w_1 \Vdash m_{r1} \ \simeq^{c_r} \ w_2 \Vdash m_{r2}}{w_1 \Vdash m_l \ |||^p_* \ m_{r1} \ \simeq^c \ w_2 \Vdash m_l \ |||^p_* \ m_{r2}} \end{array} \right.$$

$$\texttt{ap}\ c\ a = \textsc{True} \quad \frac{w_1 \Vdash m_{t1} \ \simeq^c \ w_2 \Vdash m_{t2}}{w_1 \Vdash \texttt{test}\ a\ m_{t1}\ m_{f1} \ \simeq^c \ w_2 \Vdash \texttt{test}\ a\ m_{t2}\ m_{f2}}$$

$$\texttt{ap}\ c\ a = \textsc{False} \quad \frac{w_1 \Vdash m_{f1} \ \simeq^c \ w_2 \Vdash m_{f2}}{w_1 \Vdash \texttt{test}\ a\ m_{t1}\ m_{f1} \ \simeq^c \ w_2 \Vdash \texttt{test}\ a\ m_{t2}\ m_{f2}}$$

Figure 7.2: Equivalence of world/program pairs

- If $w \Vdash m \Downarrow^c \ w_2 \Vdash m_2$, then apply Proposition 7.1.1 to show $w_1 \Vdash m_1 \Downarrow^c \ w_2 \Vdash m_2$.
- If $w \Vdash m \Uparrow^c$ then prove $w_1 \Vdash m_1 \Uparrow^c$ by contradiction by showing that if, for some $w_2 \Vdash m_2$, $w_1 \Vdash m_1 \ \Downarrow^c \ w_2 \Vdash m_2$ then $w \Vdash m \ \Downarrow^c \ w_2 \Vdash m_2$ from Proposition 7.1.1, since $w \Vdash m$ reduces to $w_1 \Vdash m_1$.

□

From the above proposition, it is clear that many world/program pairs are equivalent – if two world/program pairs can possibly reduce to one another, or reduce to a common reduct, then they are equivalent. Five of the most interesting equivalence rules can be found in Figure 7.2. These relate to the recursive constructs `>>=`, `par` and `test`, and their proofs are below.

Lemma 7.1.8. *If* $w \Vdash m \Downarrow^c \ w' \Vdash \texttt{return}\ v$ *then* $w \Vdash m \ \texttt{>>=}\ f \ \simeq^c \ w' \Vdash f\ v$.

Proof. Case analysis on whether $w \Vdash m \ \texttt{>>=}\ f$ converges:

- $w \Vdash m \texttt{>>=} f \Downarrow^c w'' \Vdash m''$, for some $w'' \Vdash m''$: apply Lemma 7.1.5. It must be the first of the two possibilities, since m reduces to value v, and we can use the lemma to prove that $w' \Vdash f\ v \Downarrow^c w'' \Vdash m''$.
- $w \Vdash m \texttt{>>=} f \Uparrow^c$: we prove by contradiction that $w' \Vdash f\ v \Uparrow^c$. If $f\ v$ did converge then, from Lemma 7.1.7, so would $m \texttt{>>=} f$.

□

Lemma 7.1.9. *If* $w \Vdash m \simeq^c w' \Vdash m'$ *then* $w \Vdash m \texttt{>>=} f \simeq^c w' \Vdash m' \texttt{>>=} f$.

Proof. Case analysis on whether $w \Vdash m$ and $w' \Vdash m'$ either both converged or both diverged:

- $w \Vdash m \Downarrow^c w_1 \Vdash m_1$ and $w' \Vdash m' \Downarrow^c w_1 \Vdash m_1$ for some $w_1 \Vdash m_1$: if m_1 is not a value, then by Lemma 7.1.6 both $w \Vdash m \texttt{>>=} f$ and $w' \Vdash m' \texttt{>>=} f$ converge to $w_1 \Vdash m_1 \texttt{>>=} f$, making them equivalent. Otherwise, if m_1 is some value $\texttt{return}\ v$ then from Lemma 7.1.8, $w \Vdash m \texttt{>>=} f \simeq^c w_1 \Vdash f\ v$ and $w' \Vdash m' \texttt{>>=} f \simeq^c w_1 \Vdash f\ v$ which, by the transitivity of $\simeq^c$ implies $w \Vdash m \texttt{>>=} f \simeq^c w' \Vdash m' \texttt{>>=} f$.
- $w \Vdash m \Uparrow^c$ and $w' \Vdash m' \Uparrow^c$: we must prove both $w \Vdash m \texttt{>>=} f \Uparrow^c$ and $w' \Vdash m' \texttt{>>=} f \Uparrow^c$, and we prove the contrapositive, namely that if either $w \Vdash m \texttt{>>=} f$ or $w' \Vdash m' \texttt{>>=} f$ *did* converge then so would $w \Vdash m$ or $w' \Vdash m'$, respectively. Apply Lemma 7.1.7 to $w \Vdash m \texttt{>>=} f$ and $w' \Vdash m' \texttt{>>=} f$ to guarantee that $w \Vdash m$ and $w' \Vdash m'$ would have to both converge to some resultant world/program pair.

□

Lemma 7.1.10. *If* $w \Vdash m_r \simeq^{c_r} w' \Vdash m'_r$ *and* $\texttt{pf}\ p\ c = (c_l, c_r)$ *then* $w \Vdash m_l \texttt{|||}^p_* m_r \simeq^c w' \Vdash m'_r \texttt{|||}^p_* m_r$.

Proof. Since $w \Vdash m_r$ and $w' \Vdash m'_r$ are equivalent, they either both diverge or both converge to the same normal form:

- If both $w \Vdash m_r$ and $w' \Vdash m'_r$ diverge: show, using Lemma 7.1.4, that this would cause both parallel programs to diverge, making them equivalent.
- If both $w \Vdash m_r$ and $w' \Vdash m'_r$ converge to some pair $w_2 \Vdash m_2$ then apply Lemma 7.1.3 to both. This lets us prove that both $w \Vdash m_l \texttt{|||}^p_* m_r \twoheadrightarrow^c w_2 \Vdash m_l \texttt{|||}^p_* m_2$ and $w' \Vdash m_l \texttt{|||}^p_* m'_r \twoheadrightarrow^c w_2 \Vdash m_l \texttt{|||}^p_* m_2$. By Proposition 7.1.3 this means both $w \Vdash m_l \texttt{|||}^p_* m_r$ and $w' \Vdash m_l \texttt{|||}^p_* m'_r$ are equivalent to $w_2 \Vdash m_l \texttt{|||}^p_* m_2$, so by the transitivity of $\simeq^c$ they, themselves, are equivalent.

□

Lemma 7.1.11. *If* $w \Vdash m_l \simeq^{c_l} w' \Vdash m'_l$ *and* $\texttt{pf}\ p\ c = (c_l, c_r)$ *then* $w \Vdash m_l \texttt{|||}^p_* m_r \simeq^c w' \Vdash m'_l \texttt{|||}^p_* m_r$.

Proof. Similar to the proof of Lemma 7.1.10. □

Lemma 7.1.12. *If* $\texttt{ap}\ a\ c = \textsc{True}$ *then* $w_1 \Vdash m_{t1} \simeq^c w_2 \Vdash m_{t2}$ *implies*

$$w_1 \Vdash \texttt{test}\ a\ m_{t1}\ m_{f1} \simeq^c w_2 \Vdash \texttt{test}\ a\ m_{t2}\ m_{f2}$$

Proof. First prove that for all w, m_t, and m_f, since $\texttt{ap}\ c\ a = \textsc{True}$, $w \Vdash \texttt{test}\ a\ m_t\ m_f \simeq^c w \Vdash m_t$. This is easy since, $w \Vdash \texttt{test}\ a\ m_t\ m_f \longrightarrow^c w \Vdash m_t$, and therefore from Proposition 7.1.3 they are equivalent. Simply apply this rule to $w_1 \Vdash m_{t1}$ and $w_2 \Vdash m_{t2}$ to show by transitivity of $\simeq^c$ that $w_1 \Vdash \texttt{test}\ a\ m_{t1}\ m_{f1} \simeq^c w_2 \Vdash \texttt{test}\ a\ m_{t2}\ m_{f2}$. □

Lemma 7.1.13. *If* $\texttt{ap}\ a\ c = \textsc{False}$ *then* $w_1 \Vdash m_{f1} \simeq^c w_2 \Vdash m_{f2}$ *implies*

$$w_1 \Vdash \texttt{test}\ a\ m_{t1}\ m_{f1} \simeq^c w_2 \Vdash \texttt{test}\ a\ m_{t2}\ m_{f2}$$

Proof. The same as the proof for Lemma 7.1.12. □

7.1.3 Deriving a big-step semantics

Having defined a useful equivalence relation $\simeq^c$ on world/program pairs, we can now quickly give a big-step semantics to Curio.

Figure 7.3 gives a "standard" big-step semantics for Curio – "standard" in the sense that it shows how the convergence of a program may somehow be inferred from the convergence of that program's sub-terms. Figure 7.4 gives the flip side of the coin, namely how the divergence of a program may be inferred.

All the rules are simple consequences of the $\simeq^c$ rules given in Figure 7.2, or of the language semantics given in Chapter 3. Here is one example: the convergence rule for the right-hand side of `par`.

Lemma 7.1.14.

$$\frac{w \Vdash m_r \Downarrow^{c_r} w' \Vdash m'_r \qquad w' \Vdash m_l \ /\!/\!/^p_* \ m'_r \Downarrow^c w'' \Vdash m''}{w \Vdash m_l \ /\!/\!/^p_* \ m_r \Downarrow^c w'' \Vdash m''}$$

Proof. If $w \Vdash m_r \Downarrow^{c_r} w' \Vdash m'_r$ then $w \Vdash m_r \simeq^{c_r} w' \Vdash m'_r$, so from the equivalence rule proved in Lemma 7.1.10, this means $w \Vdash m_l \ |||^p_* \ m_r \simeq^c w' \Vdash m_l \ |||^p_* \ m'_r$. Therefore both sides will converge to the same normal form, so $w \Vdash m_l \ |||^p_* \ m_r \Downarrow^c w'' \Vdash m''$ □

The reason why we first develop $\simeq^c$ should now be clear enough. Each equivalence result says two things: either both sides diverge, or both converge to the same resultant world/program pair. So from each equivalence relation in Figure 7.2 we get (at least) two big-step rules.

The rules formalize the following intuitions:

- $w \Vdash \texttt{return}\ v$ converges to itself.

$$\frac{}{w \Vdash \texttt{return}\ v \Downarrow^c\ w \Vdash \texttt{return}\ v}$$

$$\frac{w \Vdash m \Downarrow^c\ w_1 \Vdash m_1}{w \Vdash m\ \texttt{>>=}\ f \Downarrow^c\ w_1 \Vdash m_1\ \texttt{>>=}\ f} \quad (m_1 \text{ not a value})$$

$$\frac{w \Vdash m \Downarrow^c\ w_1 \Vdash \texttt{return}\ v \qquad w_1 \Vdash f\ v \Downarrow^c\ w_2 \Vdash m_2}{w \Vdash m\ \texttt{>>=}\ f \Downarrow^c\ w_2 \Vdash m_2}$$

$$\texttt{pf}\ p\ c = (c_l, c_r) \begin{cases} \dfrac{}{w \Vdash \texttt{return}\ v_l\ \texttt{|||}^p_*\ \texttt{return}\ v_r \Downarrow^c\ w \Vdash \texttt{return}\ v_l * v_r} \\[2ex] \dfrac{w \Vdash m_l \Downarrow^{c_l}\ w' \Vdash m'_l \qquad w' \Vdash m'_l\ \texttt{|||}^p_*\ m_r \Downarrow^c\ w'' \Vdash m''}{w \Vdash m_l\ \texttt{|||}^p_*\ m_r \Downarrow^c\ w'' \Vdash m''} \\[2ex] \dfrac{w \Vdash m_r \Downarrow^{c_r}\ w' \Vdash m'_l \qquad w' \Vdash m_l\ \texttt{|||}^p_*\ m'_r \Downarrow^c\ w'' \Vdash m''}{w \Vdash m_l\ \texttt{|||}^p_*\ m_r \Downarrow^c\ w'' \Vdash m''} \end{cases}$$

$$\texttt{ap}\ c\ a = \textsc{True} \begin{cases} \dfrac{}{w \Vdash \texttt{action}\ a \Downarrow^c\ w \Vdash \texttt{action}\ a} & \texttt{wa}\ a\ w = \textsc{True} \\[2ex] \dfrac{}{w \Vdash \texttt{action}\ a \Downarrow^c\ w_1 \Vdash \texttt{return}\ v} & \texttt{wa}\ a\ w = \textsc{False}, \texttt{af}\ a\ w = (w_1, v) \end{cases}$$

$$\frac{w \Vdash m_t \Downarrow^c\ w' \Vdash m'_t}{w \Vdash \texttt{test}\ a\ m_t\ m_f \Downarrow^c\ w' \Vdash m'_t} \quad \texttt{ap}\ c\ a = \textsc{True}$$

$$\frac{w \Vdash m_f \Downarrow^c\ w' \Vdash m'_f}{w \Vdash \texttt{test}\ a\ m_t\ m_f \Downarrow^c\ w' \Vdash m'_f} \quad \texttt{ap}\ c\ a = \textsc{False}$$

Figure 7.3: Big-step operational semantics for Curio

$$\frac{}{w \Vdash \bot \Uparrow^c} \qquad \frac{w \Vdash m \Uparrow^c}{w \Vdash m \texttt{ >>= } f \Uparrow^c} \qquad \frac{w \Vdash m \Downarrow^c \ w_1 \Vdash \texttt{return}\ v \qquad w_1 \Vdash f\ v \Uparrow^c}{w \Vdash m \texttt{ >>= } f \Uparrow^c}$$

$$\frac{}{w \Vdash \texttt{action}\ a \Uparrow^c} \ \texttt{wa}\ a\ w = \bot \qquad \frac{}{w \Vdash \texttt{action}\ a \Uparrow^c} \ \texttt{ap}\ c\ a \neq \textsc{True}$$

$$\frac{}{w \Vdash \texttt{action}\ a \Uparrow^c} \ \texttt{wa}\ a\ w = \textsc{False}, \quad \texttt{af}\ a\ w = \bot$$

$$\texttt{pf}\ p\ c = (c_l, c_r) \begin{cases} \dfrac{w \Vdash m_l \Uparrow^{c_l}}{w \Vdash m_l \mid\mid\mid^p_* m_r \Uparrow^c} & \dfrac{w \Vdash m_l \Downarrow^{c_l} \ w' \Vdash m_l' \qquad w' \Vdash m_l' \mid\mid\mid^p_* m_r \Uparrow^c}{w \Vdash m_l \mid\mid\mid^p_* m_r \Uparrow^c} \\ \dfrac{w \Vdash m_r \Uparrow^{c_r}}{w \Vdash m_l \mid\mid\mid^p_* m_r \Uparrow^c} & \dfrac{w \Vdash m_r \Downarrow^{c_r} \ w' \Vdash m_l' \qquad w' \Vdash m_l \mid\mid\mid^p_* m_r' \Uparrow^c}{w \Vdash m_l \mid\mid\mid^p_* m_r \Uparrow^c} \end{cases}$$

$$\frac{}{w \Vdash m_l \mid\mid\mid^p_* m_r \Uparrow^c} \ \texttt{pf}\ p\ c = \bot \qquad \frac{}{w \Vdash \texttt{test}\ a\ m_t\ m_f \Uparrow^c} \ \texttt{ap}\ c\ a = \bot$$

$$\frac{w \Vdash m_t \Uparrow^c}{w \Vdash \texttt{test}\ a\ m_t\ m_f \Uparrow^c} \ \texttt{ap}\ c\ a = \textsc{True} \qquad \frac{w \Vdash m_f \Uparrow^c}{w \Vdash \texttt{test}\ a\ m_t\ m_f \Uparrow^c} \ \texttt{ap}\ c\ a = \textsc{False}$$

Figure 7.4: Big-step operational semantics for Curio – divergence

- $w \Vdash m$ `>>=` f requires us to first reduce $w \Vdash m$ to normal form. If this stalls then so does m `>>=` f. If it converges to `return` v then we must then reduce $w \Vdash f\ v$ to normal form.

- $w \Vdash$ `action` a either performs the action a, converging to a resultant $w \Vdash$ `return` v, or it is stalled.

- $w \Vdash$ `test` $a\ m_t\ m_f$ reduces in the same manner as $w \Vdash m_t$ if `ap` $c\ a = \textsc{True}$, and as $w \Vdash m_f$ if `ap` c a=False.

The behaviour of the pair $w \Vdash m_l \mid\mid\mid^p_* m_r$ is somewhat trickier since the world state must always be maintained throughout and sequenced in some order. To elaborate, given a world/program pair $w \Vdash m_l \mid\mid\mid^p_* m_r$ then if we know that $w \Vdash m_l$ and $w \Vdash m_r$ have normal forms $w_l \Vdash m'_l$ and $w_r \Vdash m'_r$ respectively, we do not (at the moment) have any means of composing these two to get the resultant normal form of $w \Vdash m_l \mid\mid\mid^p_* m_r$.

There are two aspects to this. Firstly, despite our knowledge that with confluence the order in which the left- and right- hand sides are performed is irrelevant, we still have no way of merging w_l and w_r back together into a resultant world state. Secondly, even if we could "create" this resultant world state w_2, then $w_2 \Vdash m'_l \mid\mid\mid^p_* m'_r$ may not be in normal form. This is due to the existence of stalling/communication in our system. By executing m_l on its own we may have released certain actions which were stalled in world w_r – the very actions which caused m_r to originally converge to a normal form.

So the big-step rules for $w \Vdash m_l \mid\mid\mid^p_* m_r$ come in pairs – one for m_l and one for m_r. What is slightly confusing is that *it does not matter which one we choose.* Figure 7.5 demonstrates this by showing how the pair $w_0 \Vdash m_{l0} \mid\mid\mid^p_* m_{r0}$ can converge to normal form $w_F \Vdash m_{lF} \mid\mid\mid^p_* m_{rF}$. The column of reductions in the middle of the figure shows how $w_0 \Vdash m_{l0} \mid\mid\mid^p_* m_{r0}$ converges as a single program. Yet there are two different ways of understanding this convergence as the repeated application of big-step rules, as shown by the left- and right-hand sides of Figure 7.5. It depends on whether m_{l0} or m_{r0} is the first sub-program to be reduced.

Reducing m_{l0} initially	Reducing m_{r0} initially
$w_0 \Vdash m_{l0} \mid\mid\mid^p_* m_{r0}$	$w_0 \Vdash m_{l0} \mid\mid\mid^p_* m_{r0}$
$w_{l1} \Vdash m_{l1} \mid\mid\mid^p_* m_{r0}$	$w_{r1} \Vdash m_{l0} \mid\mid\mid^p_* m_{r1}$
$w_{r2} \Vdash m_{l1} \mid\mid\mid^p_* m_{r2}$	$w_{l2} \Vdash m_{l2} \mid\mid\mid^p_* m_{r1}$
$w_{l3} \Vdash m_{l3} \mid\mid\mid^p_* m_{r2}$	$w_{r3} \Vdash m_{l2} \mid\mid\mid^p_* m_{r3}$
$\vdots$	$\vdots$
$w_{lF} \Vdash m_{lF} \mid\mid\mid^p_* m_{rn}$	$w_{rF} \Vdash m_{ln} \mid\mid\mid^p_* m_{rF}$
$w_F \Vdash m_F \mid\mid\mid^p_* m_F$	$w_F \Vdash m_F \mid\mid\mid^p_* m_F$

As a comparison, there are two big-step rules for proving that $w \Vdash m$ `>>=` $f \Downarrow^c\ w_2 \Vdash m_2$, namely that $w \Vdash m$ either stalls or reduces to a value. But unlike with `par`, only one of

these two rules can hold on any occasion.

A typical example of how the behaviour in Figure 7.5 could occur is if m_{l0} and m_{r0} repeatedly communicate with each one another via two channels.

7.2 Weak program equivalence

Pure functional languages are famous for allowing equational reasoning – the substitution of denotationally equal terms within a larger program. To substantiate our claim that Curio helps us reason about I/O, it must at least allow some form of equational reasoning.

In the literature, many rigorous conditions must be imposed on program equivalence relations to guarantee their usefulness. We will consider these later, but for the moment let us begin with the basics. We have a way of identifying world/program pairs for a given context. How do we extend this to *program* equivalence for some given context?

7.2.1 Definition of weak program equivalence

An initial, naïve solution might be to define m_1 as equivalent to m_2 in context c if for all world states w, $w \Vdash m_1 \simeq^c w \Vdash m_2$. This is certainly an equivalence relation, but it is not a good one. The point of an equivalence relation is to identify programs depending on what they do, not on their syntax, or "how they look". If $w \Vdash m_1 \Downarrow^c w' \Vdash m_1'$, $w \Vdash m_2 \Downarrow^c w' \Vdash m_2'$ and both m_1' and m_2' are stalled in world w', then although they may be distinguishable, they may still always do the same thing. The $\simeq^c$ relation uses plain denotational equality in the meta-language to identify programs. As an example, in model `bfft` since `action` Rcve is denotationally different to (`action` Rcve) `>>=` $\lambda v.$ `return` v, and both may be stalled in certain world states, under the above scheme they would not be considered equivalent.

In reality, the behaviour of a program is understood in terms of how it behaves in the presence of the other arbitrary changes to the world state made by concurrent processes. We instead give the following modified, co-inductive definition of weak program equivalence which identifies two programs if, for all world states, either

- they both evaluate to exactly the same resultant world program pair in some given context.
- they both diverge.
- they both converge, after at least one reduction step, to the same world state in the given context, and, co-inductively, two equivalent programs.

We call it weak program equivalence because it is really a family of equivalence relations, one for each I/O context. Like most co-inductive definitions it is best understood intuitively in terms of observations. Proving that two programs m and m' are equivalent is really a proof

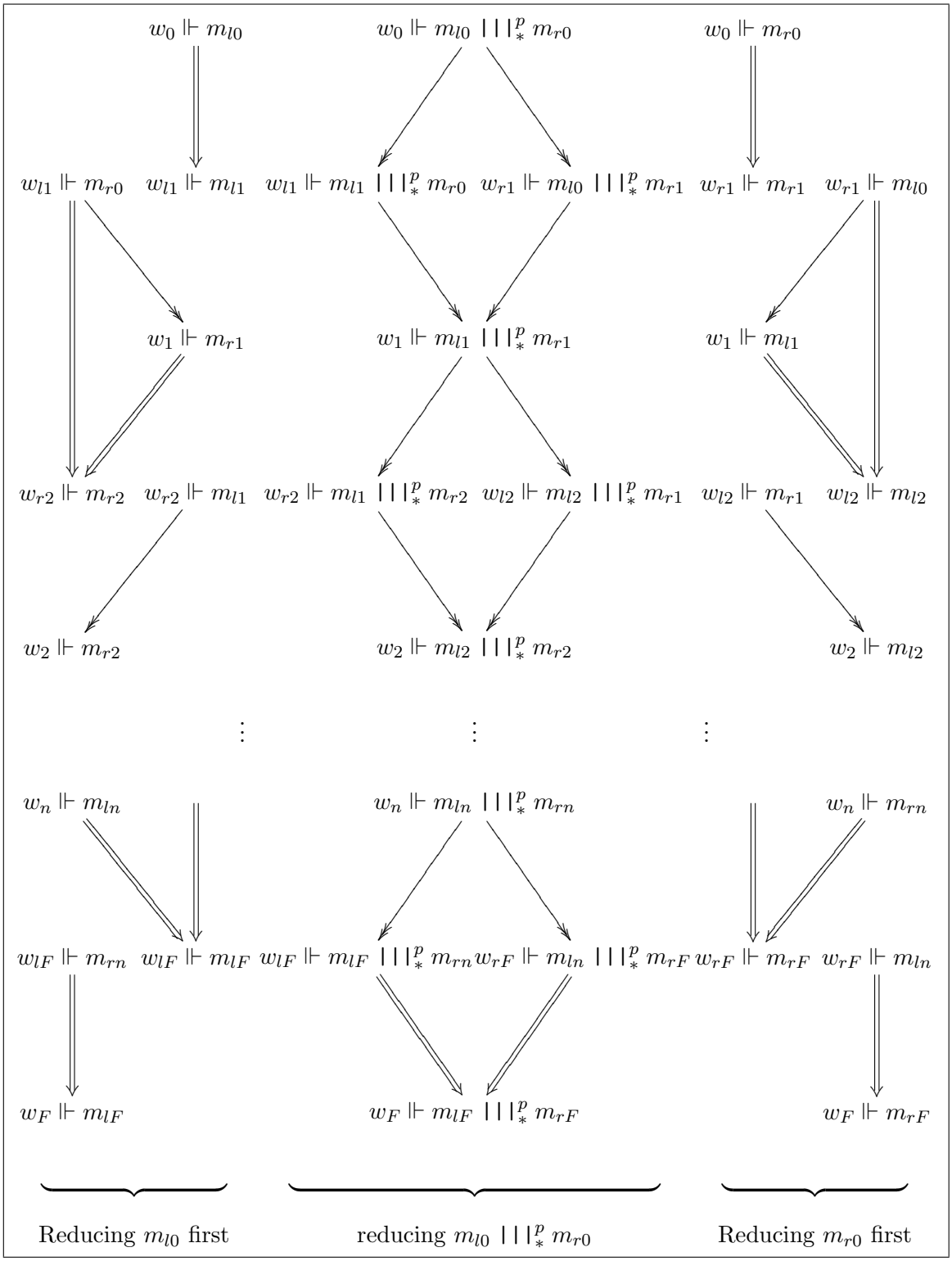


Figure 7.5: Convergence of parallel terms

that, given any infinite list of unrelated world-states $w_0, w_1, w_2, \cdots$, we can observe that the two programs produce the same, possibly finite list of output world states $w'_0, w'_1, w'_2, \cdots$

$$w_0 \Vdash m \Downarrow^c w'_0 \Vdash m_1; \quad w_1 \Vdash m_1 \Downarrow^c w'_1 \Vdash m_2; \quad w_2 \Vdash m_2 \Downarrow^c w'_2 \Vdash m_3; \cdots$$

$$w_0 \Vdash m' \Downarrow^c w'_0 \Vdash m'_1; \quad w_1 \Vdash m'_1 \Downarrow^c w'_1 \Vdash m'_2; \quad w_2 \Vdash m'_2 \Downarrow^c w'_2 \Vdash m'_3; \cdots$$

which may either continue indefinitely or until both programs simultaneously converge to the same resultant world/program pair, or simultaneously diverge.

The observation sequences of two individual concurrent programs do not compose to give the observation sequence of the single concurrent program. Consider Figure 7.5 once again, where programs m_{l0} and m_{r0} communicate with one another. The middle column shows how $m_{l0} \mid\mid\mid^p_* m_{r0}$ in world w_0 converges directly to world w_F, yet the left- and right-hand sides on their own require a number of intermediate states. So the observable effect of a program hides all the communication between its constituent sub-processes.

This co-inductive definition, for some given context c, is defined* as the greatest fixpoint of the functional f^c_{eqv} (a "functional" is a function whose argument and result are themselves functions).

$$\begin{aligned}
f^c_{eqv} \quad &: \quad (\mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathbb{B}) \to (\mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathbb{B}) \\
f^c_{eqv}\ (\cong_1)\ m_1\ m_2 \quad &\triangleq \quad \forall_{w \in \omega}. w \Vdash m_1 \simeq^c w \Vdash m_2 \ \vee \\
&\qquad \exists_{w' \in \omega}. \exists_{w'_1 \in \mathsf{Prog}\ \nu\ \alpha\ \rho}. \exists_{w'_2 \in \mathsf{Prog}\ \nu\ \alpha\ \rho}. \\
&\qquad\quad w \Vdash m_1 \overset{\geq 1}{\Downarrow^c} w' \Vdash m'_1 \ \wedge\ w \Vdash m_2 \overset{\geq 1}{\Downarrow^c} w' \Vdash m'_2 \ \wedge\ m'_1 \cong_1 m'_2
\end{aligned}$$

The elements of type $\mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathbb{B}$ form a complete lattice. The bottom and top elements are $\cong_\bot$ and $\cong_\top$ respectively, and $\sqcap$ and $\sqcup$ are the 'meet' and 'join' of the lattice:

$$\begin{aligned}
\cong_\bot &\triangleq \lambda m_1.\ \lambda m_2.\ \text{False} \\
\cong_\top &\triangleq \lambda m_1.\ \lambda m_2.\ \text{True} \\
(\cong_1) \sqcup (\cong_2) &\triangleq \lambda m_1.\ \lambda m_2.\ m_1 \cong^c_1 m_2 \ \vee\ m_1 \cong_2 m_2 \\
(\cong_1) \sqcap (\cong_2) &\triangleq \lambda m_1.\ \lambda m_2.\ m_1 \cong^c_1 m_2 \ \wedge\ m_1 \cong_2 m_2
\end{aligned}$$

This induces an ordering on all relations:

$$(\cong_1) \sqsubseteq (\cong_2) \triangleq \forall_{m_1 \in \mathsf{Prog}\ \nu\ \alpha\ \rho}. \forall_{m_2 \in \mathsf{Prog}\ \nu\ \alpha\ \rho}. m_1 \cong_1 m_2 \implies m_1 \cong_2 m_2$$

Lemma 7.2.1. *For all relations $\cong_1$ and $\cong_2$, $(\cong_1) \sqsubseteq (\cong_2)$ implies $f^c_{eqv}(\cong_1) \sqsubseteq f^c_{eqv}(\cong_2)$. In*

*This type of higher-order definition is not permitted in Sparkle, so from now on, in this chapter, we must abandon our use of a proof-assistant and work "by hand".

other words, f^c_{eqv} is monotonic for all c.

Proof. Expanding, this a proof that if for all m_3 and m_4, $m_3 \cong_1 m_4$ implies $m_3 \cong_2 m_4$, then for all m_1, m_2, $m_1(f^c_{eqv} \cong_1)m_2$ implies $m_1(f^c_{eqv} \cong_2)m_2$. In other words, given some m_1 and m_2, and world state w, if either $w \Vdash m_1 \ \simeq^c \ w \Vdash m_2$ or $w \Vdash m_1$ and $w \Vdash m_2$ converge to some $w' \Vdash m'_1$ and $w' \Vdash m'_2$ respectively and $m'_1 \cong_1 m'_2$, *then* if for all m_3, m_4 $m_3 \cong_1 m_4$ implies $m_3 \cong_2 m_4$, it can be proved that $m_1(f^c_{eqv} \cong_2)m_2$.

- $w \Vdash m_1 \ \simeq^c \ w \Vdash m_2$: regardless of $\cong_2$, $m_1(f^c_{eqv} \cong_2)m_2$ directly holds.
- $w \Vdash m_1$ and $w \Vdash m_2$ converge to some $w' \Vdash m'_1$ and $w' \Vdash m'_2$ respectively and $m'_1 \cong_1 m'_2$: we can prove that $m'_1 \cong_2 m'_2$ from the side condition, which together implies $\cong^c_2$, $m_1(f^c_{eqv} \cong_2)m_2$.

□

The monotonicity proof for f^c_{eqv} implies that its greatest fixpoint is well defined [110]. We use a greatest fixpoint, instead of a least fixpoint, to allow us to identify two infinite sequences of world state "observations".

Our program equality relation $\cong^c$ is defined as:

$$\begin{array}{lcl} \cong^c & : & \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathsf{Prog}\ \nu\ \alpha\ \rho \to \mathbb{B} \\ \cong^c & \triangleq & \nu(\cong_1).f^c_{eqv}(\cong_1) \end{array}$$

where the definition of greatest fixpoint is a standard one [40, 80]:

$$(\nu X.f(X)) \triangleq \bigsqcup\{X' \mid X' \sqsubseteq f(X')\}$$

Lemma 7.2.2. *$\cong^c$ is an equivalence relation.*

Proof.

- Reflexivity, $m_1 \ \cong^c \ m_1$: trivial from reflexivity of $\simeq^c$.
- Symmetry, $m_1 \ \cong^c \ m_2$ implies $m_2 \ \cong^c \ m_1$: if $w \Vdash m_1 \ \simeq^c \ w' \Vdash m_2$ then $w \Vdash m_2 \ \simeq^c \ w' \Vdash m_1$. If $w \Vdash m_1 \ \Downarrow^c \ w' \Vdash m'_1$, $w \Vdash m_2 \ \Downarrow^c \ w' \Vdash m'_2$ and $m'_1 \ \cong^c \ m'_2$ then $m'_2 \ \cong^c \ m'_1$ holds by co-induction and, therefore $m_2 \ \cong^c \ m_1$.
- Transitivity, $m_1 \ \cong^c \ m_2$ and $m_2 \ \cong^c \ m_3$ together imply $m_1 \ \cong^c \ m_3$: similar to proof of symmetry.

□

7.2.2 Laws of weak program equivalence

We begin with a more convenient rearrangement of the definition of $\cong^c$.

Lemma 7.2.3. *$m_1 \cong^c m_2$ is equivalent to stating that for all w either both $w \Vdash m_1 \Uparrow^c$ and $w \Vdash m_2 \Uparrow^c$ or for some w', m_1' and m_2', $w \Vdash m_1 \Downarrow^c w' \Vdash m_1'$, $w \Vdash m_2 \Downarrow^c w' \Vdash m_2'$ and $m_1' \cong^c m_2'$.*

Proof. The case in which $w \Vdash m_1 \Downarrow^c w' \Vdash m_1'$, $w \Vdash m_2 \Downarrow^c w' \Vdash m_2'$, where $m_1' = m_2'$ (i.e. both resultant programs are indistinguishable) is subsumed by $m_1' \cong^c m_2'$. □

We now use the new definition to show two fundamental equivalences. These demonstrate that if context information is treated carefully then $\cong^c$ behaves somewhat like a congruence relation, in that equivalence is preserved by primitives `>>=` and `par`.

Lemma 7.2.4. *If $m_1 \cong^c m_2$ then m_1 `>>=` $f \cong^c m_2$ `>>=` f.*

Proof. We must prove that if, for all world states w_1 either both $w_1 \Vdash m_1 \Uparrow^c$ and $w_1 \Vdash m_2 \Uparrow^c$ or $w_1 \Vdash m_1 \Downarrow^c w_1' \Vdash m_1'$, $w_1 \Vdash m_2 \Downarrow^c w_1' \Vdash m_2'$ and $m_1' \cong^c m_2'$, for some w_1', m_1' and m_2', then for all world states w either both $w \Vdash m_1$ `>>=` $f \Uparrow^c$ and $w \Vdash m_2$ `>>=` $f \Uparrow^c$ or $w \Vdash m_1$ `>>=` $f \Downarrow^c w' \Vdash m_1''$, $w \Vdash m_2$ `>>=` $f \Downarrow^c w' \Vdash m_2''$ and $m_1'' \cong^c m_2''$, for some w', m_1'' and m_2''.

- If $w \Vdash m_1 \Uparrow^c$ and $w \Vdash m_2 \Uparrow^c$ then from the big-step semantics, both $w \Vdash m_1$ `>>=` $f \Uparrow^c$ and $w \Vdash m_2$ `>>=` $f \Uparrow^c$.
- If $w \Vdash m_1 \Downarrow^c w_1' \Vdash m_1'$, $w \Vdash m_2 \Downarrow^c w_1' \Vdash m_2'$ and $m_1' \cong^c m_2'$, then if either m_1' are m_2' are identical values `return` v or both are stalled.
  - Either m_1' or m_2' are values, in which case they both must be since they are equivalent. Since $m_1' =$ `return` $v = m_2'$ then clearly m_1 `>>=` $f \cong^c m_2$ `>>=` f.
  - Neither m_1' nor m_2' are values. This, from the big-step semantics, means that $w \Vdash m_1$ `>>=` $f \Downarrow^c w_1' \Vdash m_1'$ `>>=` f and $w \Vdash m_2$ `>>=` $f \Downarrow^c w_1' \Vdash m_2'$ `>>=` f, and co-inductively we know that $m_1' \cong^c m_2'$ implies m_1' `>>=` $f \cong^c m_2'$ `>>=` f. Together this shows that m_1 `>>=` $f \cong^c m_2$ `>>=` f.

□

Lemma 7.2.5. *If `pf p c` $= (c_l, c_r)$, $m_{l1} \cong^{c_l} m_{l2}$ and $m_{r1} \cong^{c_r} m_{r2}$ then m_{l1} `|||`$^p_*$ $m_{r1} \cong^c m_{l2}$ `|||`$^p_*$ m_{r2}.*

Proof. Like with Lemma 7.2.4 we must prove that for any w, either $w \Vdash m_{l1}$ `|||`$^p_*$ $m_{r1} \Uparrow^c$ and $w \Vdash m_{l2}$ `|||`$^p_*$ $m_{r2} \Uparrow^c$, or for some w', m_1' and m_2', $w \Vdash m_{l2}$ `|||`$^p_*$ $m_{r2} \Downarrow^c w' \Vdash m_1'$, $w \Vdash m_{l2}$ `|||`$^p_*$ $m_{r2} \Downarrow^c w' \Vdash m_2''$ and $m_1' \cong^c m_2'$.

From $m_{l1} \cong^{c_l} m_{l2}$ we know that either $w \Vdash m_{l1} \Uparrow^{c_l}$ and $w \Vdash m_{l2} \Uparrow^c$ or that both converge to two equivalent programs. If it is the former then from the big-step semantics

we can prove that both $w \Vdash m_{l1} \mid\mid\mid^p_* m_{r1} \Uparrow^c$ and $w \Vdash m_{l2} \mid\mid\mid^p_* m_{r2} \Uparrow^c$. We continue with the latter case, where $w \Vdash m_{l1} \Downarrow^{c_l} w_1 \Vdash m'_{l1}$, $w \Vdash m_{l2} \Downarrow^{c_l} w_1 \Vdash m'_{l2}$ and $m'_{l1} \cong^{c_l} m'_{l2}$.

Similarly, from $m_{r1} \cong^{c_r} m_{r2}$, we can eliminate as trivial the case where both programs diverge, this time on world state w_1. We now have the following six facts

1. $w \Vdash m_{l1} \Downarrow^{c_l} w_1 \Vdash m'_{l1}$
2. $w \Vdash m_{l2} \Downarrow^{c_l} w_1 \Vdash m'_{l2}$
3. $m'_{l1} \cong^{c_l} m'_{l2}$
4. $w_1 \Vdash m_{r1} \Downarrow^{c_r} w_2 \Vdash m'_{r1}$
5. $w_1 \Vdash m_{r2} \Downarrow^{c_r} w_2 \Vdash m'_{r2}$
6. $m'_{r1} \cong^{c_r} m'_{r2}$

1 and 4 combine to give $w \Vdash m_{l1} \mid\mid\mid^p_* m_{r1} \twoheadrightarrow^c w_2 \Vdash m'_{l1} \mid\mid\mid^p_* m'_{r1}$ using Lemmas 7.1.1 and 7.1.3. Similarly, 2 and 5 combine to give $w \Vdash m_{l2} \mid\mid\mid^p_* m_{r2} \twoheadrightarrow^c w_2 \Vdash m'_{l2} \mid\mid\mid^p_* m'_{r2}$. 3 and 6 together, through co-induction, give $m'_{l1} \mid\mid\mid^p_* m'_{r1} \cong^c m'_{l2} \mid\mid\mid^p_* m'_{r2}$.

From the final, inferred equivalence we can prove that $m_{l1} \mid\mid\mid^p_* m_{r1} \cong^c m_{l2} \mid\mid\mid^p_* m_{r2}$. Both $m_{l1} \mid\mid\mid^p_* m_{r1}$ and $m_{l2} \mid\mid\mid^p_* m_{r2}$ reduce to equivalent programs, therefore they themselves must be equivalent. □

The following four lemmas are all easily proved. Since program $\bot$ always diverges, if a program is equivalent to $\bot$ in some context then it also always diverges.

Lemma 7.2.6. *If* `ap` $c\ a$ = TRUE *then* $m_{t1} \cong^c m_{t2}$ *implies* `test` $a\ m_{t1}\ m_{f1} \cong^c$ `test` $a\ m_{t2}\ m_{f2}$.

Lemma 7.2.7. *If* `ap` $c\ a$ = FALSE *then* $m_{f1} \cong^c m_{f2}$ *implies* `test` $a\ m_{t1}\ m_{f1} \cong^c$ `test` $a\ m_{t2}\ m_{f2}$.

Lemma 7.2.8. *If* `ap` $c\ a = \bot$ *then* `test` $a\ m_t\ m_f \cong^c \bot$.

Lemma 7.2.9. *If* `pf` $p\ c = \bot$ *then* m_l `|||`$^p_*$ $m_r \cong^c \bot$.

The "monad laws" [122] may also be proved. (As a result of the extra context information, these are in fact not quite as general as the actual monad laws yet, but this tiny technicality will be resolved soon).

Lemma 7.2.10. `return` v `>>=` $f \cong^c f\ v$.

Proof. Straightforward, since for every world state w, $w \Vdash$ `return` v `>>=` $f \longrightarrow^c w \Vdash f\ v$. □

Lemma 7.2.11. m `>>=` $\lambda v.$ `return` $v \cong^c m$.

Proof. Case analysis on how m reduces in some given world w.

- $w \Vdash m \Uparrow^c$: then, from the big-step semantics, $w \Vdash m \texttt{ >>= } \lambda v.\ \texttt{return}\ v \Uparrow^c$.
- $w \Vdash m \Downarrow^c\ w' \Vdash \texttt{return}\ v_1$:

  From Lemma 7.1.8, $w \Vdash m \texttt{ >>= } \lambda v.\ \texttt{return}\ v\ \simeq^c\ w' \Vdash (\lambda v.\ \texttt{return}\ v)\ v_1$ and denotationally $(\lambda v.\ \texttt{return}\ v)\ v_1$ is equal to $\texttt{return}\ v_1$. From the transitivity of $\simeq^c$, $w \Vdash m \texttt{ >>= } \lambda v.\ \texttt{return}\ v\ \simeq^c\ w \Vdash m$ because both are equivalent to $w' \Vdash \texttt{return}\ v_1$.
- $w \Vdash m \Downarrow^c\ w' \Vdash m'$, where m' is not a value: from the big-step semantics we can prove that $w \Vdash m \texttt{ >>= } \lambda v.\ \texttt{return}\ v\ \Downarrow^c\ w' \Vdash m' \texttt{ >>= } \lambda v.\ \texttt{return}\ v$, so use co-induction to prove that $m' \texttt{ >>= } \lambda v.\ \texttt{return}\ v\ \cong^c\ m'$.

□

Lemma 7.2.12. $m \texttt{ >>= } (\lambda v.\ f_1\ v \texttt{ >>= } f_2)\ \cong^c\ (m \texttt{ >>= } \lambda v.\ f_1\ v) \texttt{ >>= } f_2$.

Proof. Similar to the proof of Lemma 7.2.11. Case analysis on how m reduces in given world w. If $w \Vdash m\ \Uparrow^c$, then, from the big-step semantics, both programs diverge. If $w \Vdash m\ \Downarrow^c\ w' \Vdash \texttt{return}\ v_1$ then both world/program pairs can be proved equivalent to $w' \Vdash f_1\ v_1 \texttt{ >>= } f_2$. It $w \Vdash m\ \Downarrow^c\ w' \Vdash m'$, where m' is not a value, the ms within both expressions reduce to m's, changing world state to w'. Then use co-induction. □

7.3 Full program equivalence

Full, complete program equivalence, $\cong$, can now be defined as might be expected – one quantifies over all contexts:

$$m_1\ \cong\ m_2 \quad \triangleq \quad \forall_{c \in \varsigma}. m_1\ \cong^c\ m_2$$

7.3.1 Full equivalence rules

Many of the existing weak equivalence rules were originally proved true for all contexts, such as the monad laws. Therefore they give rise directly to full equivalence relations. The other rules, such as the congruence rules for `test` and `par` are easily proved. It is also very easy to show that $\cong$ is an equivalence relation.

These rules are shown in Figure 7.6.

7.3.2 Is full equivalence substitutive?

This is probably the single most important question one can ask about an equivalence relation over terms in a language. So, can two programs which are equivalent under $\cong$ be substituted for one another within another larger program and yield another, equivalent, larger program? (This is also known as contextual equivalence).

$$
\begin{array}{rcl}
m \texttt{ >>= } \lambda v.\ \texttt{return}\ v & \cong & m \\
\texttt{return}\ v \texttt{ >>= } f & \cong & f\ v \\
m \texttt{ >>= } (\lambda v.\ f_1\ v \texttt{ >>= } f_2) & \cong & (m \texttt{ >>= } \lambda v.\ f_1\ v) \texttt{ >>= } f_2 \\
\texttt{test}\ a\ (\texttt{test}\ a\ m_1\ m_2)\ m_3 & \cong & \texttt{test}\ a\ m_1\ m_3 \\
\texttt{test}\ a\ m_1\ (\texttt{test}\ a\ m_2\ m_3) & \cong & \texttt{test}\ a\ m_1\ m_3
\end{array}
$$

$$
\frac{m_1 \cong m_2}{m_1 \texttt{ >>= } f \cong m_2 \texttt{ >>= } f}
$$

$$
\frac{m_{l1} \cong m_{l2} \qquad m_{r1} \cong m_{r2}}{m_{l1}\ |||^p_*\ m_{r1} \cong m_{l2}\ |||^p_*\ m_{r2}}
$$

$$
\frac{m_{t1} \cong m_{t2} \qquad m_{f1} \cong m_{f2}}{\texttt{test}\ a\ m_{t1}\ m_{f1} \cong \texttt{test}\ a\ m_{t2}\ m_{f2}}
$$

Figure 7.6: Derived full equivalence rules

The answer is: a guarded "yes", subject to a few side-conditions and a little hand-waving. The issues relate largely to our meta-encoding and the higher-order nature of functional languages and monadic programming.

Monadic programming is higher-order in the sense that a program may return another monadic program. For example, consider the Haskell program:

```
getCharRet :: IO (IO Char)
getCharRet = return getChar
```

Surely if `prog1` $\cong$ `prog2` then `return prog1` $\cong$ `return prog2`, so how should our equivalence relation accommodate this – currently we just identify returned values denotationally. One option could be to define two programs `return` v_1 and `return` v_2 to be equivalent only if in some sense v_1 and v_2 are also equivalent. But in the general case, anything at all may be returned – for example a list of I/O actions or, as the following example shows, a function returning an action.

```
putStrRet :: IO (String -> IO ())
putStrRet = return putStr
```

The difficulty here is that we have abstracted away from the various building-blocks of our metalanguage syntax, such as lambda abstraction and lists, yet we all of a sudden need them back again to define our notion of program equivalence. The semantics of lazy functional languages are often given, co-inductively, by bisimulation, in which each constructor, from the outermost in, is viewed as a transition in a process calculus (see Chapter 1). We must therefore apply the same trick to our equivalence relation:

> **Side Condition 1**: We must assume that our equivalence relation is defined, coinductively, to hold over all constructors in the metalanguage.

The second problem concerns our use of the metalanguage (i.e. Core-Clean) both to implement reduction in CURIO and as the basis for most of the computational power of CURIO itself. We can prove that `return` x `>>=` $\lambda v.$ `return` $v \cong$ `return` x. Yet consider the following perfectly valid program in our metalanguage, which examines whether a program is, internally, a direct value or a monadic bind:

```
sniffer :: Prog ν α ρ → Prog ν α ρ
sniffer p = case p of
  Ret v -> Ret 0
  _     -> Ret 1
```

So even if we assume that our equivalence relation is defined coinductively under all constructors, the program

$$(\texttt{return}\ (\texttt{return}\ x\ \texttt{>>=}\ \lambda v.\ \texttt{return}\ v))\ \texttt{>>=}\ \texttt{sniffer}$$

will not behave the same as

$$\texttt{return}\ x\ \texttt{>>=}\ \texttt{sniffer}$$

since the latter will identify the returned program as being a single value but the former will not. This leads to:

> **Side Condition 2**: We must assume that functions in the metalanguage, in CURIO, do not examine the internal term-structure of any other CURIO program.

This is a standard property of lazy languages such as Haskell. Unlike in reflective languages such as LISP [70], one is not allowed to write programs which examine or rewrite other programs at runtime.

Both of these side-conditions are necessary for reasons that relate to programs which return other programs as values. Let us say we have a program m `>>=` f. If m returns some structure which contains another program m_1 then since f cannot examine m_1's internal term structure (side-condition 2) all it can do is schedule program m_1, whatever may be, to be performed at some later stage. Because other programs may be manipulated and stored in data-structures, the first side-condition states that those stored programs, since they may at some time be executed, should also be identified up to equivalence.

With these two side conditions it is safe to say that $\cong$ really is a congruence. To elaborate, in monadic I/O, all actions must eventually be "flattened" into a single sequence. So, once we know that any returned program will either be discarded or eventually executed as is, then $\cong$ becomes a congruence, because when it comes to actually performing actions we will be working solely with CURIO's primitives once again.

To get, formally, from a congruence to substitutive equality traditionally requires some extra work. Howe, in [51], describes the relationship between various forms of equality for a general class of lazy computational systems (bisimulation, when it is a congruence, and when it corresponds to contextual equivalence). Milner's Context Lemma [72] has also been used for similar purposes.

We should note that this informality is also present in other research in this area. In [40, 39] Gordon uses Howe's method [51] to prove formally that his equality via bisimulation is substitutive. This, however, requires an entire operational semantics for a small, minimal functional language. Peyton Jones [89, 88], on the other hand, uses the same denotational approach, modelling I/O in a full-blown functional language, and is therefore forced to skim over any formal details regarding notions of program equivalence since the metalanguage syntax is no longer explicit.

It is also worth noting that none of the rules established earlier are either wrong or lacking rigour. It is merely with the above side-conditions that we can see equivalence as spreading to all aspects of the metalanguage, as well as the new language primitives in Curio.

7.4 Equivalence proofs for I/O actions

Thus far in this chapter we have avoided any mention of I/O actions, instead proving more sterile results relating to program flow-control constructs. There is nothing stopping us from doing this. In this section we give a few examples.

7.4.1 Some small proofs

To start with, in model $\mathfrak{ivar}$, it may be shown that

$$\begin{aligned}
\texttt{action}\ (\textsc{WriteI}\ i_1)\ \texttt{>>}\ \texttt{action}\ (\textsc{WriteI}\ i_2) &\cong \texttt{action}\ (\textsc{WriteI}\ i_2) \\
\texttt{action}\ (\textsc{WriteI}\ i)\ \texttt{>>}\ \texttt{action}\ \textsc{ReadI} &\cong \texttt{action}\ (\textsc{WriteI}\ i)\ \texttt{>>}\ \texttt{return}\ i \\
\texttt{action}\ \textsc{ReadI}\ \texttt{>>}\ \texttt{action}\ \textsc{ReadI} &\cong \texttt{action}\ \textsc{ReadI}
\end{aligned}$$

These are not hard to prove. They hold because, for any of the three possible contexts WriteC, ReadC and NoneC, if one side fails because the context is too weak to permit a particular action then the other side fails simultaneously. Also, no actions stall in model $\mathfrak{ivar}$, so we merely need to prove $\simeq^c$ for all world states and contexts c. But the following equivalence must be annotated with context information, because in context NoneC the left-hand side would fail.

$$\texttt{action}\ \textsc{ReadI}\ \texttt{>>=}\ \lambda i.\ \texttt{action}\ (\textsc{WriteI}\ i) \cong^{\textsc{WriteC}} \texttt{return}\ 0$$

7.4.2 A larger proof

Reasoning about programs which stall is of importance. We shall now prove that in model $\mathfrak{bfft}$

$$\texttt{rcveSum}\ 0 \ \cong^{c_R}\ \texttt{rcveList} \texttt{ >>= } \lambda l.\ \texttt{return}\ (\texttt{sum}\ l)$$

where c_R = SubC False, the context which permits receiving. Here, `rcveSum` receives integers from the buffer until a negative number is encountered, returning the sum of the received integers, and `rcveList` returns a list of received integers, stopping when a negative integer is found. The left-hand program computes the sum of the received integers incrementally, whereas the right-hand program performs the sum only at the very end.

```
rcveSum :: Int → Prog_bfft Int
rcveSum s = action Rcve >>= \i ->
               if (i<0) then (return s) else (rcveSum (s+i))

rcveList :: Prog_bfft [Int]
rcveList = action Rcve >>= \i ->
  if (i<0) then (return []) else (rcveList >>= \l -> return (i:l))
```

The world state in $\mathfrak{bfft}$ is a list of integers. It may therefore, by case analysis, either be $\bot$, [] or (i:is) for some i and is. If it is of the form (i:is) and $i \geq 0$ then according to the operational semantics, execution continues as normal:

$$\left.\begin{array}{c}(i{:}is) \Vdash \texttt{rcveSum}\ s \longrightarrow\!\!\!\!\!\rightarrow^{c_R} is \Vdash \texttt{rcveSum}\ (s+i) \\ (i{:}is) \Vdash \texttt{rcveList} \longrightarrow\!\!\!\!\!\rightarrow^{c_R} is \Vdash \texttt{rcveList} \texttt{ >>= } \lambda l.\ \texttt{return}\ (i{:}l)\end{array}\right\} i \geq 0$$

If $i < 0$, however, then execution terminates with a return value.

$$\left.\begin{array}{c}(i{:}is) \Vdash \texttt{rcveSum}\ s \Downarrow^{c_R}\ is \Vdash \texttt{return}\ s \\ (i{:}is) \Vdash \texttt{rcveList} \Downarrow^{c_R}\ is \Vdash \texttt{return}\ []\end{array}\right\} i < 0$$

If world state is [] then both `rcveSum` and `rcveList` become stalled.

$$[] \Vdash \texttt{rcveSum}\ s \Downarrow^{c_R}\ [] \Vdash \texttt{rcveSum}\ s \qquad [] \Vdash \texttt{rcveList} \Downarrow^{c_R}\ [] \Vdash \texttt{rcveList}$$

If world state is $\bot$ then both programs diverge:

$$\bot \Vdash \texttt{rcveSum}\ s \Uparrow^{c_R} \qquad \bot \Vdash \texttt{rcveList} \Uparrow^{c_R}$$

Lemma 7.4.1. *Regardless of m,*

$$(m \texttt{ >>= } \lambda l_1.\ \texttt{return}\ (i_1{:}l_1)) \texttt{ >>= } \lambda l_2.\ \texttt{return}\ (i_2{:}l_2) \ \cong\ m \texttt{ >>= } \lambda l.\ \texttt{return}\ (i_2{:}(i_1{:}l))$$

Proof. A direct application of the monad laws and the definition of `sum`. □

The following two lemmas may now be proved by inducting over the length of the world state list and using Lemma 7.4.1.

Lemma 7.4.2. *If list 'is' is a possibly empty list* $[i_0, i_1, \cdots, i_j]$*, and all* $i_0, i_1, \cdots, i_j$ *are non-negative then*

$$is \Vdash \mathit{rcveSum}\ s \Downarrow^{c_R} [] \Vdash \mathit{rcveSum}\ (s + i_0 + i_1 + \cdots + i_j)$$

and

$$is \Vdash \mathit{rcveList} \Downarrow^{c_R} [] \Vdash \mathit{rcveList} \mathbin{>\!\!>\!=} \lambda l.\ \mathit{return}\ (is +\!\!+ l)$$

Lemma 7.4.3. *If 'is' is a non-empty list* $[i_0, i_1, \cdots, i_j]$*,* i_k *is negative, and* $i_0, \cdots, i_{k-1}$ *are all non-negative then*

$$is \Vdash \mathit{rcveSum}\ s \Downarrow^{c_R} [i_{k+1}, \cdots, i_j] \Vdash \mathit{return}\ (s + i_0 + i_1 + \cdots + i_{k-1})$$

and

$$is \Vdash \mathit{rcveList} \Downarrow^{c_R} [i_{k+1}, \cdots, i_j] \Vdash \mathit{return}\ [i_0, i_1, \cdots, i_{k-1}]$$

Theorem 7.4.1.

$$\mathit{rcveSum}\ s \cong^{c_R} \mathit{rcveList} \mathbin{>\!\!>\!=} \lambda l.\ \mathit{return}\ (s + \mathit{sum}\ l)$$

Proof. Let is be the world state. If $is = \bot$ then both programs diverge. Otherwise either apply Lemma 7.4.2 or Lemma 7.4.3 depending on whether a negative integer is an element of is.

If not, both programs become stalled:

$$is \Vdash \texttt{rcveSum}\ s \Downarrow^{c_R} [] \Vdash \texttt{rcveSum}\ (s + i_0 + i_1 + \cdots + i_j)$$
$$is \Vdash \texttt{rcveList} \Downarrow^{c_R} [] \Vdash \texttt{rcveList} \mathbin{>\!\!>\!=} \lambda l.\ \texttt{return}\ (is +\!\!+ l)$$

Using the monad laws and the big-step semantics the second fact may be arranged to prove that:

$$is \Vdash \texttt{rcveList} \mathbin{>\!\!>\!=} \lambda l.\ \texttt{return}\ (s + \texttt{sum}\ l) \Downarrow^{c_R} [] \Vdash \texttt{rcveList} \mathbin{>\!\!>\!=} \lambda l.\ \texttt{return}\ (\texttt{sum}\ is) + (\texttt{sum}\ l)$$

Now since both programs result in the same final world state, co-induction may be used to show that $\texttt{rcveSum}\ (\texttt{sum}\ is) \cong^{c_R} \texttt{rcveList} \mathbin{>\!\!>\!=} \lambda l.\ \texttt{return}\ ((\texttt{sum}\ is) + \texttt{sum}\ l)$

If, on the other hand, there exists some negative integer i_k, then

$$is \Vdash \texttt{rcveSum}\ s \Downarrow^{c_R} [i_{k+1}, \cdots, i_j] \Vdash \texttt{return}\ (s + i_0 + i_1 + \cdots + i_{k-1})$$
$$is \Vdash \texttt{rcveList} \Downarrow^{c_R} [i_{k+1}, \cdots, i_j] \Vdash \texttt{return}\ [i_0, i_1, \cdots, i_{k-1}]$$

The second fact may be arranged using the monad laws and the definition of `sum` to show that:

$$is \Vdash \texttt{rcveList} \texttt{ >>= } \lambda l.\, \texttt{return}\ (s + \texttt{sum}\ l) \Downarrow^{c_R} \ [i_{k+1}, \cdots, i_j] \Vdash \texttt{return}\ (\texttt{sum}\ [i_0, i_1, \cdots, i_{k-1}])$$

Therefore for the given world state both programs are identical □

This theorem above can be easily specialised to show that

$$\texttt{rcveSum}\ 0 \ \cong^{c_R} \ \texttt{rcveList} \texttt{ >>= } \lambda l.\, \texttt{return}\ (\texttt{sum}\ l)$$

7.4.3 Non-termination

Before the introduction of equivalence relations non-termination was mostly simple. If two programs did not terminate, either by never converging to normal form or through non-termination in the meta-language, then they were considered equal. Our co-inductive equivalence relation has muddied the water somewhat, however.

As an example, consider the programs `sends` and `rcves` which, respectively, perform Send and Rcve indefinitely:

```
sends :: Prog_bfft ()
sends = action (Send 3) >> sends

rcves :: Prog_bfft ()
rcves = action Rcve >> rcves
```

Although `sends` $\cong \bot$, it is not true that `rcves` $\cong \bot$. In every world state w and context c, $w \Vdash$ `sends` $\Uparrow^c$. `rcves`, however, may stall. So given any infinite sequence of world states, `sends` will always diverge on the very first one, but `rcves` will usually for each consecutive world state consume every character from the buffer and then stall.

If we view correct termination as "converging to a value" then this is awkward, because both `sends` and `rcves` are equally divergent. If, on the other hand, we view convergence to some stalled program as a legitimate way for a program to terminate then it makes perfect sense to distinguish them.

7.5 Large-scale I/O proofs

We have shown how simple programs may be proved equivalent for various example I/O models but what about model `io` defined in Chapter 6? Formal reasoning about the effects of I/O actions is not particularly common in the literature. Butterfield [14] performed a small case-study comparing the ease of formal reasoning about I/O in C, Haskell and Clean. The author followed this with a larger case-study [32] in which a simplified functional version of the UNIX `make` tool was proved correct. The only other example we can find of I/O

proofs is associated with the House [45] functional operating system, which lets one state and verify simple properties of monadic programs which interact with program hardware.

At the moment there are various issues which may make "real world" equivalence proofs in Curio, though possible, rather cumbersome. These are:

- World state, actions and contexts are domain-based.
- Full equality is perhaps too strong a condition.
- Contexts and the way in which they are split do not obey algebraic properties.

Some of these require slight extensions to pre-condition $\mathsf{PRE}_{\mathfrak{s}}$, which guarantees confluence of program execution but very little else. These possible extensions, defined in Figure 7.7, will be explained in the following pages and tackled head-on in Chapter 8.

7.5.1 Domain-based I/O models

Our world state, actions and I/O contexts are all domains with a $\perp$ element. This is not a major difficulty but it is unnecessary and sometimes highly unintuitive. For example the rule

$$\texttt{test}\ a\ m\ m\ \cong\ m$$

does not hold, because for action $a = \perp$, or for context $\perp$, it is almost certain that the left-hand side will fail. In many cases, also, an equivalence relation involving I/O actions will not hold simply because of the possibility that the world state is undefined. For example, a program which adds 1 to an integer, and then subtracts 1 from that integer should be the same as a program which does nothing.

A domain-theoretic style of proof-assistant was required to reason about actual programs, but the models themselves would probably be more suited to a constructive, strongly-normalising approach (that is, the sort of constructive type theory [19] used as the basis for the Coq [114] theorem prover).

7.5.2 Hybrid program equivalence

We have defined two program equivalence relations, one which quantifies over all contexts and one which only holds for a given, specific context. For real program fragments, neither may be entirely satisfactory. Often in the presence of concurrency, one will want to prove two programs equivalent with respect to a *minimum* set of permissions.

Define the relation $\cong^{c\sqsubseteq}$, meaning "equal for all contexts with at least the permissions of context c":

$$m_1\ \cong^{c\sqsubseteq}\ m_2 \quad \triangleq \quad \forall_{c' \in \varsigma}.c \sqsubseteq_{\mathfrak{s}} c' \Longrightarrow m_1\ \cong^{c'}\ m_2$$

This hybrid relation obeys all the rules common to $\cong^c$ and $\cong$, and the following extra rules:

$$
\begin{array}{rcl}
\mathsf{mon} & \triangleq & \forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\forall_{c'\in\varsigma}.\forall_{c'_l\in\varsigma}.\forall_{c'_r\in\varsigma}. \\
& & \quad \mathtt{pf}\ p\ c = (c_l\,,c_r) \Longrightarrow \mathtt{pf}\ p\ c' = (c'_l\,,c'_r) \Longrightarrow c \sqsubseteq_{\mathfrak{s}} c' \Longrightarrow c_r \sqsubseteq_{\mathfrak{s}} c'_r\ \wedge\ c_l \sqsubseteq_{\mathfrak{s}} c'_l \\
\mathsf{lft} & \triangleq & \exists_{\mathtt{lft}\in\rho}.\forall_{c\in\varsigma}.\exists_{c_r\in\varsigma}.\mathtt{pf}\ \mathtt{lft}\ c = (c\,,c_r) \\
\mathsf{sym} & \triangleq & \exists_{\mathtt{sym}\in\rho\to\rho}.\forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\mathtt{pf}\ p\ c = (c_l\,,c_r) \Longrightarrow \mathtt{pf}\ (\mathtt{sym}\ p)\ c = (c_r\,,c_l) \\
\mathsf{asl} & \triangleq & \exists_{\mathtt{asl}\in(\rho,\rho)\to(\rho,\rho)}.\forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_1\in\varsigma}.\forall_{c_2\in\varsigma}.\forall_{c_3\in\varsigma}. \\
& & \quad (\exists_{c_T\in\varsigma}.\mathtt{pf}\ p_1\ c = (c_T\,,c_3)\ \wedge\ \mathtt{pf}\ p_2\ c_T = (c_1\,,c_2)) \Longrightarrow \exists_{p_1\in\rho}.\exists_{p_2\in\rho}. \\
& & \qquad \mathtt{asl}\ (p_1\,,p_2) = (p'_1\,,p'_2)\ \wedge\ (\exists_{c_T\in\varsigma}.\mathtt{pf}\ p'_1\ c = (c_1\,,c_T)\ \wedge\ \mathtt{pf}\ p'_2\ c_T = (c_2\,,c_3))
\end{array}
$$

Figure 7.7: Additional pre-conditions

- If $\mathtt{ap}\ c\ a = \textsc{True}$ then $\mathtt{test}\ a\ m_t\ m_f\ \cong^{c\sqsubseteq}\ m_t$.
- If additional pre-condition mon holds, as defined in Figure 7.7, then

$$\frac{m_{l1}\ \cong^{c_l\sqsubseteq}\ m_{l2} \qquad m_{r1}\ \cong^{c_r\sqsubseteq}\ m_{r2}}{m_{l1}\ |||^{p}_{*}\ m_{r1}\ \cong^{c\sqsubseteq}\ m_{l2}\ |||^{p}_{*}\ m_{r2}}\quad \mathtt{pf}\ p\ c = (c_l\,,c_r)$$

Once an equivalence result of this form is established then one can wrap both programs in a series of `test` commands to make them fully equivalent – if the program has the requested permissions then it goes ahead, otherwise it performs some sort of common exception handler.

With a stronger `test` command this could even be automated as a rule. Imagine a richer command `testC`. $\mathtt{testC}\ c\ m_t\ m_f$ would check to see if *all* the actions permitted by context c are allowed by the current context. If this existed then the following rule would hold, linking $\cong^{c\sqsubseteq}$ to full equality, $\cong$:

$$\frac{m_{t1}\ \cong^{c\sqsubseteq}\ m_{t2}}{\mathtt{testC}\ c\ m_{t1}\ m_f\ \cong\ \mathtt{testC}\ c\ m_{t2}\ m_f}$$

7.5.3 Manipulating concurrent programs

One awkward problem with Curio is that I/O contexts obey effectively no algebraic properties, and for this reason we don't possess even the most basic laws for manipulating programs in the presence of concurrency. Figure 7.7 contains three additional pre-conditions which we might like `pf` to obey: sym, lft and asl.

If sym holds, then we can flip the syntactic left-to-right order of concurrency:

$$m_l\ |||^{p}_{*}\ m_r\ \cong\ m_r\ |||^{\mathtt{sym}\ p}_{\mathtt{flip}\ (*)}\ m_l$$

If lft holds, then one can give all permissions to the left-hand side when performing

concurrency:

$$m \cong m \;|||^{\texttt{lft}}_{\texttt{fst}}\; (\texttt{return}\; v)$$

Finally, if asl holds, and $\texttt{asl}\ (p_1,p_2) = (p_1',p_2')$, then[†]

$$m_1 \;|||^{p_1}_{*_1}\; (m_2 \;|||^{p_2}_{*_2}\; m_3) \cong (m_1 \;|||^{p_2'}_{\texttt{higher}\ *_1\ *_2}\; m_2) \;|||^{p_1'}_{\$}\; m_3$$

This is a kind of "associativity" condition for `par`. The original functions $*_1$ and $*_2$ are replaced by `higher` $*_1$ $*_2$ and `$`. The higher-order function `higher` $*_1$ $*_2$, when applied to the return values of m_1 and m_2, creates a function, and this function is then applied directly to the result of m_3 using the application operator, `$`.

$\texttt{higher} :: (\delta \to \gamma \to \epsilon) \to (\alpha \to \beta \to \gamma) \to \delta \to \alpha \to (\beta \to \epsilon)$

```
higher f1 f2 d a = \b -> f1 d (f2 a b)
```

As an example, model term obeys all of these extra conditions. In the term model, one may distribute permissions to `stdin` and `stdout` whichever way one wants just as long as access to each individual resource is single threaded:

```
sym (TCxt bp bg)                  = TCxt bg bp
lft                               = TCxt True True
asl (TCxt bp1 bg1, TCxt bp2 bg2) =
        (TCxt bp1 bg1, TCxt (bp1 || bp2) (bg1 || bg2))
```

The definition of `asl` above is not an obvious one, and in fact it can be far from obvious whether it holds for any given I/O model. In general, these additional properties suggest the need for a more appropriate mathematical structure for I/O contexts, and that is the goal of the next chapter.

7.6 Chapter summary

In this chapter we first gave a big-step semantics for Curio, and then used this to give a co-inductive definition of equivalence for Curio programs. This notion of equivalence in effect stated that equivalent programs modify world state in identical ways, and also respond to arbitrary changes in world state identically. This equivalence relation was shown to obey the monad laws and be substitutive, and we gave a few sample proofs. The chapter concluded with a discussion of some of the current limitations to formal reasoning caused by the definition of our I/O model structure.

Some of these limitations were attributed to the weakness of our precondition $\mathsf{PRE}_{\mathfrak{s}}$ – it guarantees confluence, but little else. In Chapter 8 we design a new structure which describes I/O contexts more suitably, thereby resolving many of these difficulties.

[†]One could also imagine mirror-image versions of asl and lft. It is quite easy to show that if sym holds these may be derived from asl and lft.

Chapter 8

A lattice-theoretic approach to I/O contexts

The Curio language definition and the specification for I/O models, given in Chapters 2 and 3, were designed mainly with a view to giving a language implementation which permitted a rigorous, machine-checkable confluence proof. However, although the pre-condition guarantees confluence, the actual way in which contexts are split is left entirely unspecified. This makes for a general confluence condition, applicable to many situations, but if the type ρ, used to specify the splitting of contexts, does not obey any algebraic properties then it will be impossible to prove many interesting program equivalences. This was hinted at in Chapter 7, when we gave a program

$$m_1 \mid\mid\mid^{p_1}_{*_1} (m_2 \mid\mid\mid^{p_2}_{*_2} m_3)$$

and asked how we would determine p'_1, p'_2, $*'_1$ and $*'_2$, if any existed, such that

$$m_1 \mid\mid\mid^{p_1}_{*_1} (m_2 \mid\mid\mid^{p_2}_{*_2} m_3) \;\cong\; (m_1 \mid\mid\mid^{p'_2}_{*'_1} m_2) \mid\mid\mid^{p'_1}_{*'_2} m_3$$

We gave a solution to this for $*'_1$ and $*'_2$, but not for p'_1 and p'_2 on account of our lack of understanding of what p_1 and p_2 mean. We shall refer to this throughout as the "associativity problem".

We therefore need to find a worthy candidate for ρ and `pf` to replace the existing underspecified ones. Ideally, this should

- Infer the most general sub-contexts in a given situation.
- Allow the user to specify clearly the permissions he/she would like to give to the sub-programs and behave "reasonably" when those permissions cannot be granted.
- Obey algebraic properties.

This is the overall goal of this chapter. To do it we must first re-examine I/O contexts themselves, so as to describe $c_1 \diamondsuit_{\mathfrak{s}} c_2$ at a clearer level of abstraction. We start in Section 8.1 by considering an API at a high-level, concentrating solely on whether two actions are order independent (commute with one another), ignoring everything to do with the state-transformer and world model. Treating contexts as sets of actions, this gives rise in Section 8.2 to what we call a "maximal lattice" whose elements are contexts. We call it "maximal" since only the largest "useful" contexts are contained therein – as a quick example, if an action a commutes with all actions then there is no reason for any context to forbid a. Each context c has an associated unique inverse context $\neg c$ which is the largest set of actions which may be safely performed in parallel with c. This is used in Section 8.3 to provide a direct translation of $\mathsf{PRE}_{\mathfrak{s}}$ and other additional axiomatic properties, and give two possible candidates for the `pf` function which obey some of them. Unfortunately, we show how we do not yet have a solution to the associativity problem, and that something more elaborate will be necessary. Section 8.4 concludes, comparing maximal lattices to other algebraic structures in the literature.

8.1 An axiomatisation of I/O contexts

We begin by stripping I/O models and I/O contexts down to their bare essentials. This mostly means abstracting away from the behaviour of actions and treating whether two actions can be performed concurrently (or commute with one another) at an axiomatic level. Whether two actions commute with one another now becomes the starting point, rather than a result proved from a state-transformer.

8.1.1 Definition

Let us assume the following:

$$\begin{array}{rcl} a, a_1, a_2, \cdots & : & A \\ c, c_1, c_2, \cdots & : & \mathcal{P}A \\ \odot & : & A \to A \to \mathbb{B} \\ a_1 \odot a_2 & \Leftrightarrow & a_2 \odot a_1 \\ \neg & : & \mathcal{P}A \to \mathcal{P}A \\ \neg c & \triangleq & \{a \in A \mid \forall_{a_1 \in c}. a \odot a_1\} \end{array}$$

The set A represents a fixed, finite*, global set of actions. Variables $a, a_1, a_2 \cdots$ are individual actions in A, and c, c_1, c_2 $\cdots$ denote subsets of a A, which can be thought of

*We could probably generalise this to infinite sets with a little effort, but the simpler, finite case suffices for all examples in this chapter.

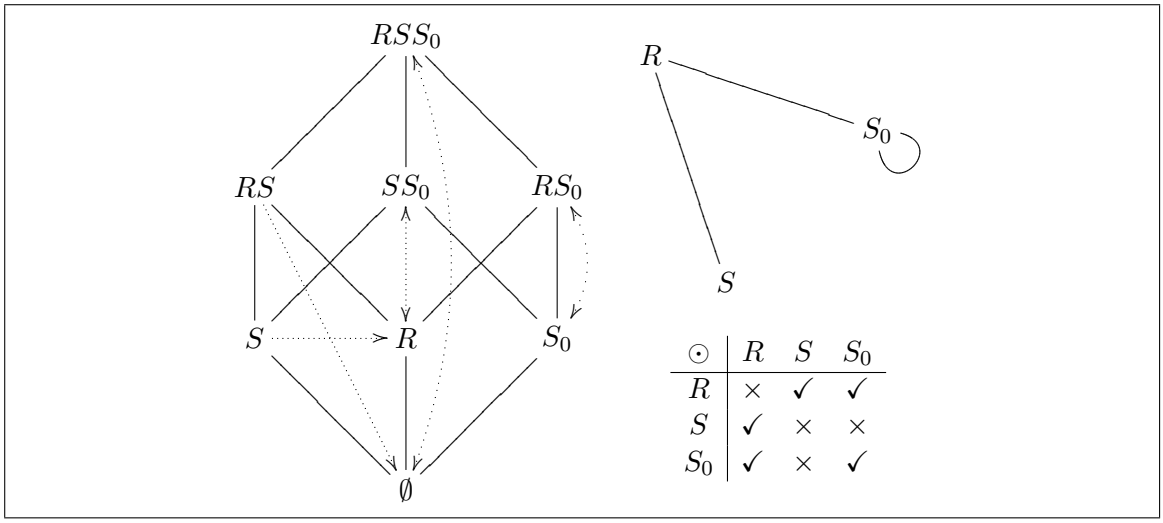


Figure 8.1: I/O model "RSS_0"

as contexts. $a_1 \odot a_2$ indicates whether a_1 commutes with a_2, and we assume that $\odot$ is a symmetric relation.

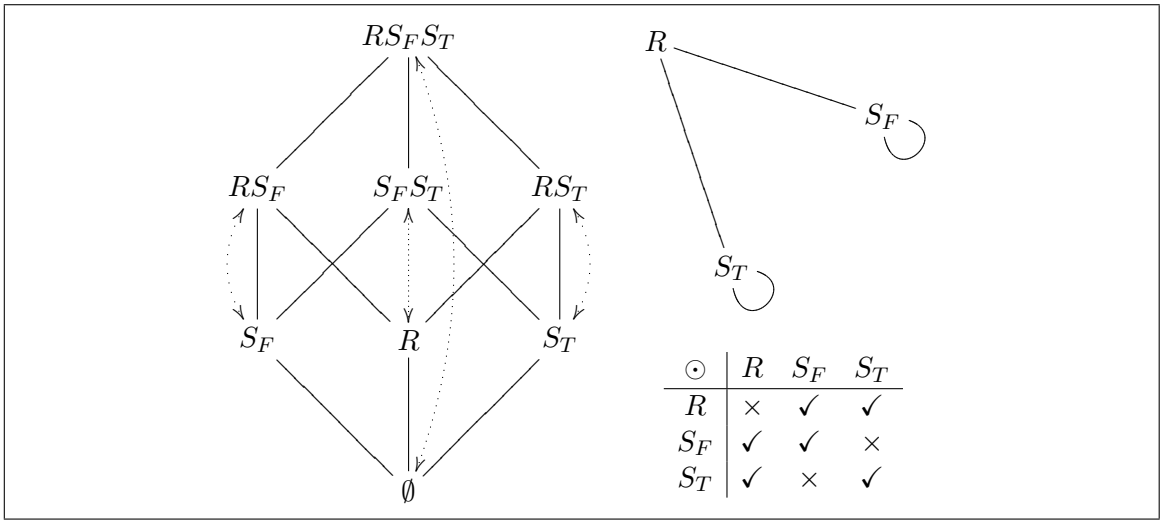
The unary operator $\neg$ gives the **inverse** of a context. For a context c, $\neg c$ denotes the largest subset of A such that all actions $a \in c$ commute with all actions $a_1 \in \neg c$. So if a program is running in context c and another program is running, concurrently, in context $\neg c$ (or a subset of $\neg c$), then execution will still be deterministic.

We also assume the existence of the standard set operators $\cup$ ("union") and $\cap$ ("intersection"), and a unary complement operator $\sim$ which obeys the property $\sim c = A \backslash c$. It is well-known [26] that $\langle \cap, \cup, \sim, A, \emptyset \rangle$ forms a Boolean algebra, a special kind of lattice, obeying the following properties:

- Associativity and commutativity of $\cap$ and $\cup$.
- Idempotence. $a \cap a = a = a \cup a$
- Distributivity. $a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$ and $a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$
- Absorption. $a \cap (b \cup a) = a = a \cup (b \cap a)$
- de Morgan laws. $\sim(c_1 \cup c_2) = \sim c_1 \cap \sim c_2$ and $\sim(c_1 \cap c_2) = \sim c_1 \cup \sim c_2$
- Complements. $\sim\sim c = c$, $c \cup \sim c = A$, $c \cap \sim c = \emptyset$, $\sim\emptyset = A$ and $\sim A = \emptyset$

Figures 8.1 and 8.2 contain small examples which will be used throughout this chapter. Although they are slightly contrived, they are non-trivial and have a good practical interpretation.

In model RSS_0, in Figure 8.1, the world state is a communication buffer and one can read/write integers to and from it. However, as well as permitting concurrent "receive"s and "send"s, we also want to allow multiple processes to concurrently send the value 0 –



$\odot$	R	S_F	S_T
R	×	✓	✓
S_F	✓	✓	×
S_T	✓	×	✓

Figure 8.2: I/O model "RS_FS_T"

something permitted *only* if 0s can be sent. This is to guarantee that the receiving process cannot tell the difference. We identify three basic actions, or equivalence classes of actions:

- R: receive the next integer from the buffer.
- S: send any non-zero integer along the buffer. (i.e. the equivalence class of all "send" actions which communicate non-zero integers).
- S_0: send 0 along the buffer.

The relation $\odot$ for these actions can be seen in both table and graph form in Figure 8.1. R commutes with both S and S_0 but not with itself, and S_0 commutes with itself – and that is all. Also included is the Boolean Algebra of subsets of $\{R, S, S_0\}$. The behaviour of the function $\neg$ is indicated with a dotted arrow and, to save space, we shorten the element names by removing braces and commas. For example, the sets $\{R, S_0\}$ and $\{R, S, S_0\}$ are shortened to RS_0 and RSS_0, but $\emptyset$ remains unchanged.

Model RS_FS_T in Figure 8.2 is only slightly different. Once again the world state is a communication buffer, but only Boolean values may be sent along it. The three actions are:

- R: receive the next Boolean value.
- S_F: send "False" along the buffer.
- S_T: send "True" along the buffer.

A receive commutes with any send, S_F commutes with itself and S_T also commutes with itself.

8.1.2 Relation to I/O models

All of this information may be extracted directly from a standard (confluent) I/O model:

- The set A can be taken to be the elements of the domain α, or a suitable finite equivalence class of those elements. To make the presentation clearer we will usually ignore uninteresting elements of α, such as $\bot$.

- Then define the $\odot$ operator as either of the following: (it should be clear that both of these are symmetric definitions)

$$\begin{aligned} a_1 \odot a_2 &\triangleq a_1 \mid\mid\mid_{\mathfrak{s}} a_2 \ \wedge\ \mathsf{ally}_{\mathfrak{s}}(a_1, a_2) \ \wedge\ \mathsf{ally}_{\mathfrak{s}}(a_2, a_1) \\ a_1 \odot a_2 &\triangleq \exists_{c_1 \in \varsigma}.\exists_{c_2 \in \varsigma}.\mathtt{ap}\ a_1\ c_1 = \textsc{True} \ \wedge\ \mathtt{ap}\ a_2\ c_2 = \textsc{True} \ \wedge \\ &\qquad (\exists_{p \in \rho}.\exists_{c \in \varsigma}.\mathtt{pf}\ p\ c = (c_1, c_2) \ \vee\ \mathtt{pf}\ p\ c = (c_2, c_1)) \end{aligned}$$

The difference between the first and the second is slightly subtle. The first definition defines two actions to commute if they really do commute in the model; the second is weaker and only defines actions to commute if the I/O model allows them to be performed concurrently (and since the I/O model is confluent this will imply that they really do commute).

We can also prove a "completeness" result for this particular axiomatisation. It is reasonable to ask whether all symmetric relations $\odot$ on a set A have some corresponding state-transformer on a world state? Can we be certain if the relation $\odot$ is extracted from an I/O model that it will just be symmetric and not constrained in any further way?

The following simple construction shows that this is the case. We show it to be true for all finite sets A, but it should generalise easily to countably infinite sets.

Lemma 8.1.1. *For any finite set A and symmetric relation $\odot : A \to A \to \mathbb{B}$ there exists a state-transformer $\mathtt{af} :: A \to \omega \to (\omega, \nu)$ such that $a_1 \mid\mid\mid a_2$ (under $\mathtt{af}$) if and only if $a_1 \odot a_2$.*

Proof. Let n be the number of elements in A. We refer to these n actions as $a_1, a_2, \cdots, a_n$. Now, define the types ν and ω as follows:

$$\begin{aligned} \nu &\triangleq \mathsf{Int} \\ \omega &\triangleq (\underbrace{\mathsf{Int}, \mathsf{Int}, \cdots, \mathsf{Int}}_{n \text{ times}}) \end{aligned}$$

The world state is n integer counters, and each action a_j returns the value of the jth counter and increments zero or more of the other counters. The state-transformer is defined in terms of $\mathtt{incr}_{\odot} : \mathsf{Int} \to \omega \to \omega$. The function $\mathtt{incr}_{\odot}\ j$ defines the effect of action a_j on world state: for any counter i_k, if $a_j \odot a_k$ does *not* hold, then i_k is incremented. Action a_j

always returns the (initial) value i_j of the jth counter.

$$\begin{array}{rcl}
\texttt{wa}\ a\ w & \triangleq & \textsc{False} \\
\texttt{incr}_\odot & : & \mathsf{Int} \to \omega \to \omega \\
\texttt{incr}_\odot\ j\ (i_1,i_2,\cdots,i_n) & \triangleq & (\text{if } a_j \odot a_1 \text{ then } i_1 \text{ else } i_1+1, \\
 & & \text{if } a_j \odot a_2 \text{ then } i_2 \text{ else } i_2+1, \\
 & & \vdots \\
 & & \text{if } a_j \odot a_n \text{ then } i_n \text{ else } i_n+1) \\
\texttt{af}\ a_j\ (i_1,i_2,\cdots,i_n) & \triangleq & (\texttt{incr}_\odot\ j\ (i_1,i_2,\cdots,i_n),i_j)
\end{array}$$

It can be shown easily that $\texttt{incr}_\odot\ j\ \circ\ \texttt{incr}_\odot\ k = \texttt{incr}_\odot\ k\ \circ\ \texttt{incr}_\odot\ j$. This is because adding either 0 or 1 to a counter is order independent, and because the effect of an action on a particular counter (i.e. whether it increments it) is constant. So if we were to disregard return values then all actions would commute with one another – it is the actions' return values alone which may be affected by ordering.

Assuming a family of projection functions $\pi_1, \pi_2, \cdots \pi_n$ which return the 1st, 2nd, $\cdots$ nth elements of an n-tuple, then the following fact is also easily shown:

$$\pi_j(\texttt{incr}_\odot\ k\ w) = \begin{cases} \pi_j(w) & ,\text{if } a_j \odot a_k \\ \pi_j(w)+1 & ,\text{if } a_j \not\odot a_k \end{cases}$$

Say the world state is $(i_1,i_2,\cdots,i_n)$ and we wish to perform both actions a_j and a_k. If $a_j \odot a_k$ then neither actions will affect i_j and i_k, so regardless of the ordering, a_j and a_k will return i_j and i_k respectively. If $a_j \not\odot a_k$, then performing a_j and then a_k will give return values i_j and i_k+1 respectively, whereas performing a_k and then a_j will give return values i_k and i_j+1.

We want to prove that $a_j \mid\mid\mid a_k \iff a_j \odot a_k$. The left-hand side is:

$$\begin{array}{c}
\forall_{w\in\omega}.\forall_{w_2\in\omega}.\texttt{wa}\ a_l\ w = \textsc{False} \wedge \texttt{wa}\ a_r\ w = \textsc{False} \Longrightarrow \forall_{v_j\in\nu}.\forall_{v_k\in\nu}. \\
(\exists_{w_1\in\omega}.\texttt{af}\ a_j\ w = (w_1,v_j) \wedge \texttt{af}\ a_k\ w_1 = (w_2,v_k)) \\
\Longleftrightarrow \\
(\exists_{w_1\in\omega}.\texttt{af}\ a_k\ w = (w_1,v_k) \wedge \texttt{af}\ a_j\ w_1 = (w_2,v_j))
\end{array}$$

No actions are stalled ($\texttt{wa}\ a\ w = \textsc{False}$ for all a, w), and since the resultant world state is always order independent, this becomes

$$\begin{array}{c}
\forall_{w\in\omega}.\forall_{v_j\in\nu}.\forall_{v_k\in\nu}. \\
(v_j = \pi_j(w) \wedge v_k = \pi_k(\texttt{incr}_\odot\ j\ w)) \Longleftrightarrow (v_k = \pi_k(w) \wedge v_j = \pi_j(\texttt{incr}_\odot\ k\ w))
\end{array}$$

If $a_j \odot a_k$, then this is a proof that $(v_j = \pi_j(w) \wedge v_k = \pi_k(w)) \iff (v_k = \pi_k(w) \wedge v_j = \pi_j(w)))$, which is always true, and therefore corresponds to the truth value of $a_j \odot a_k$. If $a_j \not\odot a_k$ then it is a proof of $(v_j = \pi_j(w) \wedge v_k = \pi_k(w)+1) \iff (v_k = \pi_k(w) \wedge v_j = \pi_j(w)+1))$, which is false, corresponding to the truth value of $a_j \odot a_k$. □

As an example, if the above construction was used with the RSS_0 model, it would result in the following state-transformer:

$$\begin{aligned}
\texttt{af}\ R\ (i_R, i_S, i_{S_0}) &= ((i_R + 1, i_S, i_{S_0}), i_R) \\
\texttt{af}\ S\ (i_R, i_S, i_{S_0}) &= ((i_R, i_S + 1, i_{S_0} + 1), i_S) \\
\texttt{af}\ S_0\ (i_R, i_S, i_{S_0}) &= ((i_R, i_S + 1, i_{S_0}), i_{S_0})
\end{aligned}$$

8.1.3 Properties of inversion

The remainder of this section is spent proving a collection of properties about $\neg$.

Proposition 8.1.1. $\neg\emptyset = A$

Proof. $\neg\emptyset = \{a \in A \mid \forall_{a_1 \in \emptyset}. a \odot a_1\} = \{a \in A \mid \text{True}\} = A$. □

(It should be noted that $\neg A$ is the set of actions which commute with all other actions, and therefore is not necessarily equal to $\emptyset$).

Proposition 8.1.2. $c \subseteq \neg\neg c$

Proof. Expanding the definition, $\neg\neg c = \{a \in A \mid \forall_{a_1 \in A}. (\forall_{a_2 \in c}. a_1 \odot a_2) \implies a \odot a_1\}$. We must therefore prove that if $a \in c$, then for all a_1 such that a_1 commutes with all elements of c, $a \odot a_1$. This holds, because since every a_1 commutes with all elements of c, a_1 also commutes with a, being an element of c. □

Proposition 8.1.3. $\neg(c_1 \cup c_2) = \neg c_1 \cap \neg c_2$

Proof.

$$\begin{aligned}
\neg(c_1 \cup c_2) &= \{a \in A \mid \forall_{a_1 \in c_1 \cup c_2}. a \odot a_1\} \\
&= \{a \in A \mid (\forall_{a_1 \in c_1}. a \odot a_1) \ \wedge\ (\forall_{a_1 \in c_2}. a \odot a_1)\} \\
&= \{a \in A \mid (\forall_{a_1 \in c_1}. a \odot a_1)\} \ \cap\ \{a \in A \mid (\forall_{a_1 \in c_2}. a \odot a_1)\} \\
&= \neg c_1 \cap \neg c_2
\end{aligned}$$

□

Proposition 8.1.4. *If* $c_1 \subseteq c_2$ *then* $\neg c_2 \subseteq \neg c_1$

Proof. If $c_1 \subseteq c_2$ then $c_1 \cup c_2 = c_2$, which implies $\neg(c_1 \cup c_2) = \neg c_2$. Using Proposition 8.1.3, prove $\neg c_1 \cap \neg c_2 = \neg c_2$, which means that $\neg c_2 \subseteq \neg c_1$. □

Lemma 8.1.2. *If* $c_1 \subseteq c_2$ *then* $\neg\neg c_1 \subseteq \neg\neg c_2$

Proof. Apply Proposition 8.1.4 twice successively: $c_1 \subseteq c_2$ implies $\neg c_2 \subseteq \neg c_1$, which implies $\neg\neg c_1 \subseteq \neg\neg c_2$. □

Lemma 8.1.3. $\neg c = \neg\neg\neg c$

Proof. Prove two separate inequalities:

- $\neg c \subseteq \neg\neg\neg c$: A direct consequence of Proposition 8.1.2 ($(\neg c) \subseteq \neg\neg(\neg c)$)
- $\neg\neg\neg c \subseteq \neg c$: From Proposition 8.1.2 we know that $c \subseteq \neg\neg c$, so apply Proposition 8.1.4 to prove that $\neg\neg\neg c \subseteq \neg c$.

□

8.2 Defining a maximal lattice

In this section we define a sub-lattice of the existing Boolean algebra of I/O contexts. The elements of this sub-lattice are those contexts c such that $c = \neg c_1$, for some c_1.

The basic insight behind this manoeuvre is that we want to identify those "core" contexts c for which $\neg\neg c = c$, and if $c = \neg c_1$ for some c_1 then this is true. To elaborate, given a context c, $\neg c$ denotes the actions that may be run in parallel with the actions in c. However, $\neg c$ also denotes the actions which may be run in parallel with $\neg\neg c$ since, from Lemma 8.1.3, $\neg(\neg\neg c) = \neg c$. Therefore if $c \subset \neg\neg c$ then context c has no real purpose on its own, unless one wants to deliberately constrain the actions one performs, because by replacing it with $\neg\neg c$ one then expands the set of actions it is capable of performing without restricting the set of actions that may be executed concurrently with it.

As an example, in the model RSS_0 there are two contexts such that $c \subset \neg\neg c$: S and RS. The double inversion of these are SS_0 and RSS_0 respectively. In both cases there is nothing to be gained by forbidding S_0, since if one can send any non-zero integer then it will always be safe to send 0 also, so contexts S or RS should be excluded.

To be absolutely clear about whether we are talking about a context c in $\mathcal{P}A$ or one in the sub-lattice we annotate context variables with $+$ to indicated that the context is the inversion of some other context. Contexts in the sub-lattice are of the form $c^+, c_1^+, c_2^+, \cdots$. In fact, c^+ is really just shorthand for $\neg c$, and the two will be used interchangeably.

That the set of contexts $\{c \in \mathcal{P}A \mid c = \neg\neg c\}$ forms a sub-lattice is an easy consequence of Tarski's Fixpoint Theorem [110]. From Lemma 8.1.2, applied twice, we know that $c_1 \subseteq c_2$ implies $\neg\neg c_1 \subseteq \neg\neg c_2$, which means that the function $\lambda c.\neg\neg c$ is monotonic. Therefore, the set of c such that $c = \neg\neg c$ (the set of fixpoints of $\lambda c.\neg\neg c$) forms a sub-lattice of the existing

lattice/Boolean algebra. Tarski's original proof was not constructive, merely stating that a sub-lattice exists, so we prove the details in full giving a precise definition of $\sqcup$ and $\sqcap$. A constructive proof of Tarksi's theorem was given by Cousot and Cousot in [21], but it is considerably more complex, so we just prove the result directly.

Theorem 8.2.1. *Given a set A and a symmetric relation $\odot : A \to A \to \mathbb{B}$, then $\sqcap$ and $\sqcup$ form a lattice over the set $\mathcal{A}$, with zero $\mathbb{0}$ and unit $\mathbb{1}$, where*

$$\begin{array}{rcl}
\mathcal{A} & \triangleq & \{c \in \mathcal{P}A | \exists_{c_1 \in \mathcal{P}A}. c = \neg c_1\} \quad (\subseteq \mathcal{P}A) \\
c_1^+ \sqcup c_2^+ & \triangleq & \neg\neg(c_1^+ \cup c_2^+) \\
c_1^+ \sqcap c_2^+ & \triangleq & c_1^+ \cap c_2^+ \\
\mathbb{1} & \triangleq & \neg\emptyset \\
\mathbb{0} & \triangleq & \neg\neg\emptyset \\
\neg & : & \mathcal{P}A \to \mathcal{P}A \\
\neg c & \triangleq & \{a \in A \mid \forall_{a_1 \in c}. a \odot a_1\}
\end{array}$$

and $\neg$ is a de Morgan involution on the lattice (namely, $\neg\neg c^+ = c^+$, $\neg(c_1^+ \sqcup c_2^+) = \neg c_1^+ \sqcap \neg c_2^+$, $\neg(c_1^+ \sqcap c_2^+) = \neg c_1^+ \sqcup \neg c_2^+$ and $\neg\mathbb{0} = \mathbb{1}$).

Proof. Firstly, are the types of $\sqcup, \sqcap, \neg, \mathbb{0}, \mathbb{1}$ correct? That is, when given some $\neg c_1$, $\neg c_2$ as parameters do they then return $\neg c_3$, for some c_3? This is clearly true for $\neg$, and $\sqcup$, $\mathbb{0}$ and $\mathbb{1}$ are type-correct by definition. $\sqcap$ can be shown to be type-correct using Proposition 8.1.3: $c_1^+ \sqcap c_2^+ = \neg c_1 \cap \neg c_2 = \neg(c_1 \cup c_2)$.

We prove now that $\neg$ is a de Morgan involution.

- Inversion: $\neg\neg c_1^+ = \neg\neg\neg c_1 =$ (Lemma 8.1.3) $= \neg c_1 = c_1^+$.
- de Morgan 1: $\neg(c_1^+ \sqcup c_2^+) = \neg\neg\neg(c_1^+ \cup c_2^+) =$ (Lemma 8.1.3) $= \neg(c_1^+ \cup c_2^+) =$ (Proposition 8.1.3) $= \neg c_1^+ \cap \neg c_2^+ = \neg c_1^+ \sqcap \neg c_2^+$.
- de Morgan 2: $\neg(c_1^+ \sqcap c_2^+) =$ (Inversion) $= \neg(\neg\neg c_1^+ \sqcap \neg\neg c_2^+) =$ (de Morgan 1) $= \neg\neg(\neg c_1^+ \sqcup \neg c_2^+) =$ (Inversion) $= \neg c_1^+ \sqcup \neg c_2^+$.
- Zero Inversion: $\neg\mathbb{0} = \neg\neg\neg\emptyset = \neg\emptyset = \mathbb{1}$

Since $\sqcap = \cap$, it is immediate that $\sqcap$ is commutative, associative and idempotent, and because $\sqcap$ is the de Morgan dual of $\sqcup$ under $\neg$, the commutativity, associativity and idempotence of $\sqcup$ follows directly from that of $\sqcap$:

- Commutativity of $\sqcup$: $c_1^+ \sqcup c_2^+ =$ (Inversion) $= \neg\neg(c_1^+ \sqcup c_2^+) =$ (de Morgan 1) $= \neg(\neg c_1^+ \sqcap \neg c_2^+) =$ (Commutativity of $\sqcap$) $= \neg(\neg c_2^+ \sqcap \neg c_1^+) =$ (de Morgan 1) $= \neg\neg(c_2^+ \sqcup c_1^+) =$ (Inversion) $= c_2^+ \sqcup c_1^+$

- Associativity of $\sqcup$: $(c_1^+ \sqcup c_2^+) \sqcup c_3^+ =$ (Inversion) $\neg\neg((c_1^+ \sqcup c_2^+) \sqcup c_3^+) =$ (de Morgan 1) $= \neg((\neg c_1^+ \sqcap \neg c_2^+) \sqcap \neg c_3^+) =$ (Associativity of $\sqcap$) $= \neg(\neg c_1^+ \sqcap (\neg c_2^+ \sqcap \neg c_3^+)) =$ (de Morgan 2) $= \neg\neg c_1^+ \sqcup (\neg\neg c_2^+ \sqcup \neg\neg c_3^+) =$ (Inversion) $= c_1^+ \sqcup (c_2^+ \sqcup c_3^+)$

- Idempotence of $\sqcup$: $c^+ \sqcup c^+ =$ (Inversion) $= \neg\neg(c^+ \sqcup c^+) =$ (de Morgan 1) $= \neg(\neg c^+ \sqcap \neg c^+) =$ (Idempotence of $\sqcap$) $= \neg(\neg c^+) =$ (Inversion) $= c^+$.

Next the absorption rules need to be proved:

- Absorption 1: $c_1^+ \sqcup (c_1^+ \sqcap c_2^+) = \neg\neg(c_1^+ \cup (c_1^+ \cap c_2^+)) = \neg\neg c_1^+ = c_1^+$

- Absorption 2: $c_1^+ \sqcap (c_1^+ \sqcup c_2^+) = \neg\neg(c_1^+ \sqcap (c_1^+ \sqcup c_2^+)) =$ (de Morgan 1 & 2) $= \neg(\neg c_1^+ \sqcup (\neg c_1^+ \sqcap \neg c_2^+)) =$ (Absorption 1) $= \neg(\neg c_1^+) = c_1^+$

Finally:

- $\mathbb{1} \sqcup c^+ = \neg\neg(\neg\emptyset \cup c^+) = \neg\neg(A \cup c^+) = \neg\neg A = A = \neg\emptyset = \mathbb{1}$

- $\mathbb{1} \sqcap c^+ = \neg\emptyset \cap c^+ = A \cap c^+ = c^+$

- $\mathbb{0} \sqcup c^+ = \neg\neg(\mathbb{0} \sqcup c^+) = \neg(\mathbb{1} \sqcap \neg c^+) = \neg(\neg c^+) = c^+$

- $\mathbb{0} \sqcap c^+ = \neg\neg(\mathbb{0} \sqcap c^+) = \neg(\mathbb{1} \sqcup \neg c^+) = \neg\mathbb{1} = \mathbb{0}$

□

Such a structure extracted from a set A and relation $\odot$ is called the **maximal lattice of** $\odot$, or just, in short, a maximal lattice. A maximal lattice is said to be of **order** n if its underlying set A has n elements.

Maximal lattices are not necessarily distributive. Instead, the following weaker rules hold:

Proposition 8.2.1.

$$\begin{array}{lrcl} (i) & (c_1^+ \sqcap c_2^+) \sqcup (c_1^+ \sqcap c_3^+) & \subseteq & c_1^+ \sqcap (c_2^+ \sqcup c_3^+) \\ (ii) & c_1^+ \sqcup (c_2^+ \sqcap c_3^+) & \subseteq & (c_1^+ \sqcup c_2^+) \sqcap (c_1^+ \sqcup c_3^+) \end{array}$$

Proof. To prove (i), first rewrite the left-hand side, making use of the distributive law for $\cup$ and $\cap$:

$$\begin{aligned} (c_1^+ \sqcap c_2^+) \sqcup (c_1^+ \sqcap c_3^+) &= \neg\neg((\neg c_1 \cap c_2^+) \cup (\neg c_1 \cap c_3^+)) \\ &= \neg(\neg(\neg c_1 \cap c_2^+) \cap \neg(\neg c_1 \cap c_3^+)) \\ &= \neg((\neg\neg c_1 \cup \neg c_2^+) \cap (\neg\neg c_1 \cup \neg c_3^+)) \\ &= \neg(\neg\neg c_1 \cup (\neg c_2^+ \cap \neg c_3^+)) \end{aligned}$$

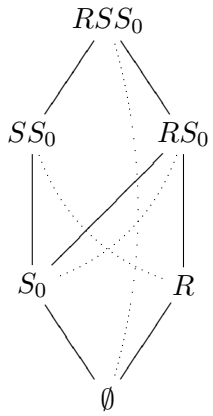
The right-hand side also rewrites as follows:

$$\begin{aligned} c_1^+ \sqcap (c_2^+ \sqcup c_3^+) &= \neg c_1 \cap \neg\neg(c_2^+ \cup c_3^+) \\ &= \neg(c_1 \cup \neg(c_2^+ \cup c_3^+)) \\ &= \neg(c_1 \cup (\neg c_2^+ \cap \neg c_3^+)) \end{aligned}$$

Now make the simple observation that $c_1 \subseteq \neg\neg c_1$. Therefore, for any c_4, $c_1 \cup c_4 \subseteq \neg\neg c_1 \cup c_4$, which implies that $\neg(\neg\neg c_1 \cup c_4) \subseteq \neg(c_1 \cup c_4)$. So to prove inequality (i), substitute $\neg c_2^+ \cap \neg c_3^+$ for c_4. Its dual (ii) follows without difficulty. □

8.2.1 Examples

The maximal lattice for the RSS_0 model is as below. The elements RS and S have been removed, leaving the six useful contexts. As with the previous diagram a dotted line indicates the inversion $\neg c^+$ of a context c^+, but since $\neg\neg c^+ = c^+$ in maximal lattices there is no longer any need to show a direction using an arrowhead.



(The maximal lattice for the RS_FS_T model is identical to the original Boolean algebra – each context is the inversion of some other context).

Figure 8.3 contains the six different maximal lattices (up to isomorphism) which are order 2. In the figure, each lattice is accompanied on its left-hand side by the undirected graph of its $\odot$ relation.

These six lattices are

- $XY1$: no concurrency is permitted whatsoever, so the only contexts are XY (everything) and $\emptyset$ (nothing).

- $XY2$: X commutes with itself, but Y doesn't commute with anything. A context X is necessary because without it one could not exploit this concurrency – $\emptyset$ allows nothing and XY allows too many actions to permit (useful) concurrency. But context Y is not necessary, because if a context allows action Y then it might as well allow X too.

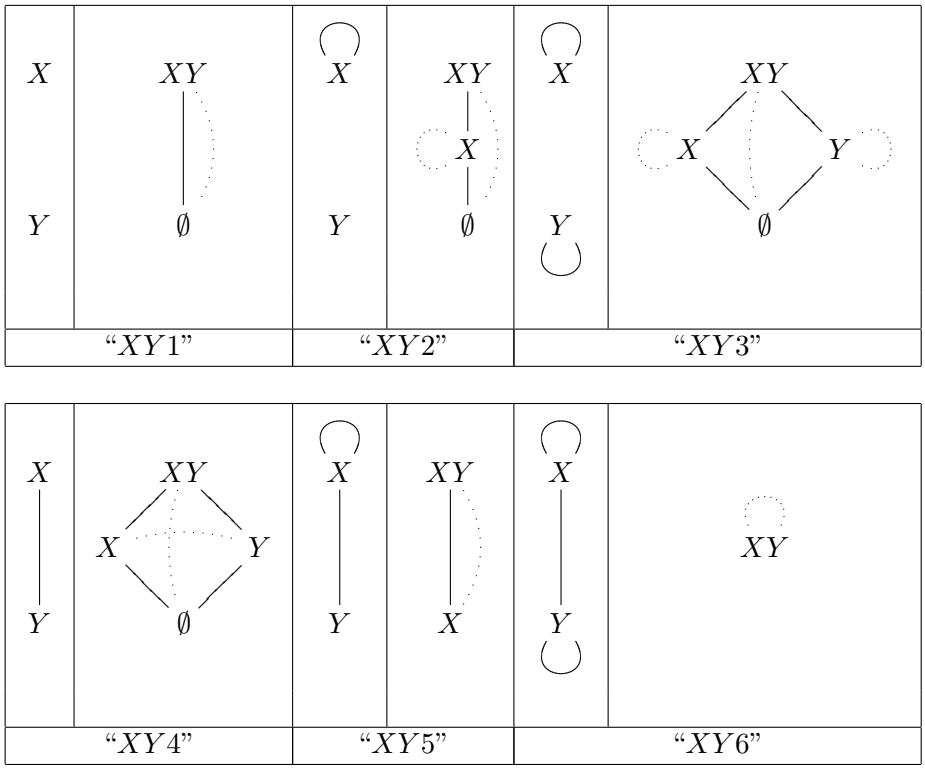


Figure 8.3: The six maximal lattices of order 2

- $XY3$, $XY4$: we need to include contexts which represent all four subsets of $\{X, Y\}$.
- $XY5$: action X commutes with all actions (including itself), so there is no reason to forbid it, but action Y must be run in a single threaded fashion. XY forms the top element of the lattice, and X the bottom element.
- $XY6$: all actions commute with all actions, so concurrency is not constrained in any way. Only one context is needed, XY, and this permits all actions.

Figure 8.4 contains two examples of non-distributive maximal lattices. It is a well-known theoretical result (referred to as the $\mathbf{M}_3 - \mathbf{N}_5$ Theorem in [26]) that all non-distributive lattices contain a sub-lattice whose diagram is like either the upper or lower lattice in Figure 8.4. Taking the upper lattice: $Y \sqcup (X \sqcap Z) = Y \sqcup \emptyset = Y$, but $(Y \sqcup X) \sqcap (Y \sqcup Z) = XYZ \sqcap XYZ = XYZ$. Therefore $Y \sqcup (X \sqcap Z) \neq (Y \sqcup X) \sqcap (Y \sqcup Z)$.

These examples also show why we cannot define $\sqcup$ to be $\cup$. In both lattices, $X \sqcup Y = XYZ$ but $X \cup Y = XY$, and XY is not an element of the maximal lattice. It is for this reason that, $c_1^+ \sqcup c_2^+$ is defined to be $\neg\neg(c_1^+ \cup c_2^+)$. Clearly, if $\sqcup$ is indistinguishable from $\cup$ for a maximal lattice then that lattice is distributive, since $\cup$ and $\cap$ distribute over each other in a Boolean Algebra.

Figure 8.5 contains the maximal lattices of four of the five small examples given in Chapter 2, namely lock, ivar, istr and term. The model for bffr was omitted since it is

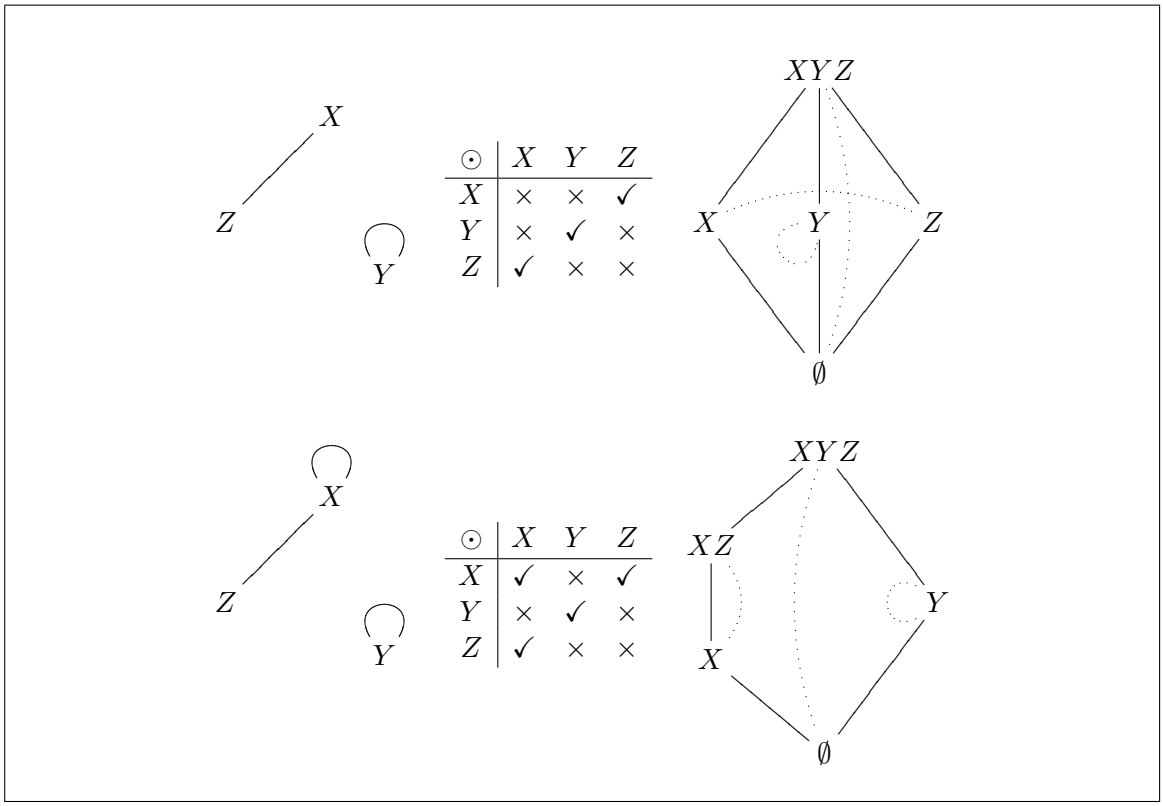


Figure 8.4: Two non-distributive maximal lattices

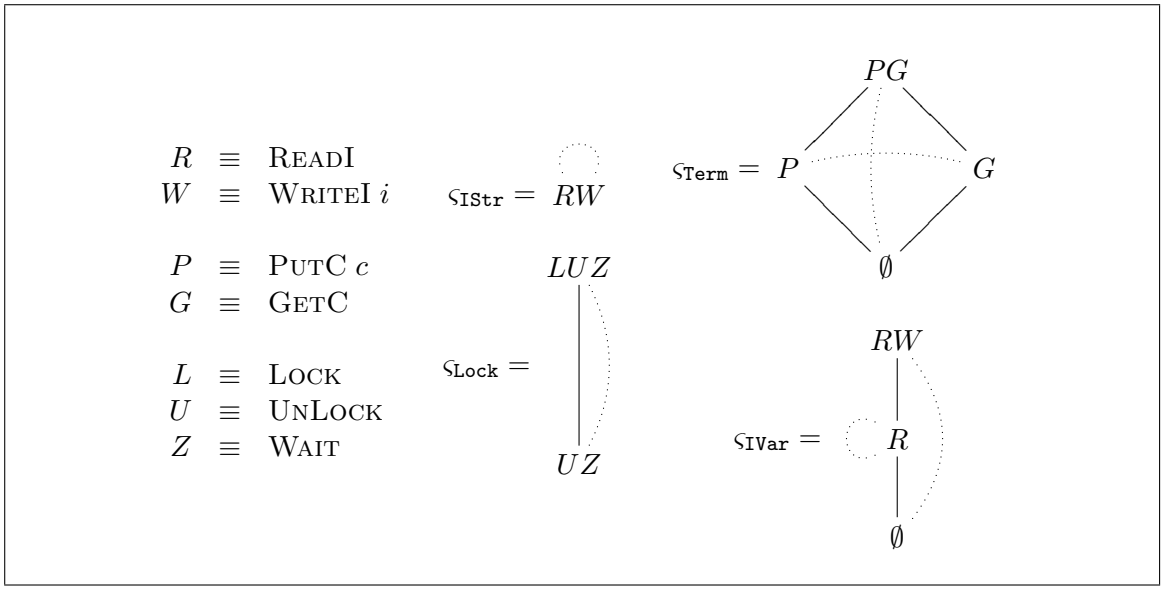
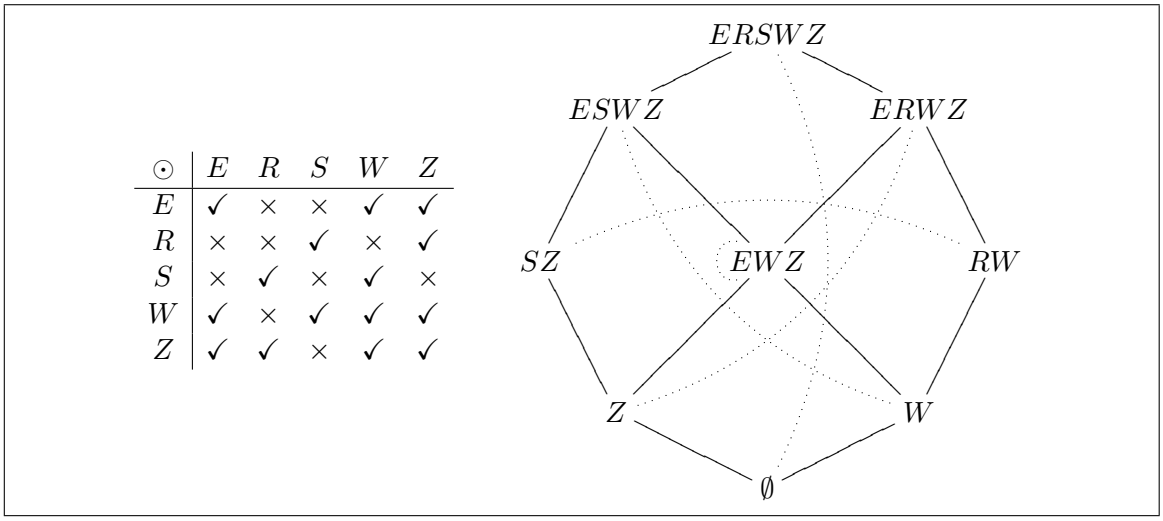


Figure 8.5: Maximal lattices for example I/O models



$\odot$	E	R	S	W	Z
E	✓	×	×	✓	✓
R	×	×	✓	×	✓
S	×	✓	×	✓	×
W	✓	×	✓	✓	✓
Z	✓	✓	×	✓	✓

Figure 8.6: Maximal lattice for model "$ERSWZ$"

isomorphic to that of $\mathfrak{term}$.

The final and largest maximal lattice example which we shall give can be found in Figure 8.6. This is a send/receive buffer model with three additional actions, E, W and Z.

- R: receive an item from the buffer.
- S: send an item along the buffer.
- E: return whether the buffer is empty.
- W: wait until the buffer is non-empty.
- Z: wait until the buffer is empty.

R and S commute with one another, as do R and Z, and S and W. Additionally, E, W and Z all commute with one another. The resultant maximal lattice is far from obvious yet it may be computed directly from the table of $\odot$. The contexts SZ and RW are the traditional "sender" and "receiver" contexts, yet the lattice also exposes further opportunities for concurrency.

8.3 Mechanisms for splitting contexts

We are now able to rewrite $\mathsf{PRE}_{\mathfrak{s}}$ and all the other axiomatic properties of I/O models solely using operations on a maximal lattice. By abstracting I/O models in this way we can give some candidates for a general-purpose algorithm for the splitting of contexts.

The original definition of $\mathsf{PRE}_{\mathfrak{s}}$ from Chapter 2 is as follows:

$$\mathsf{PRE}_{\mathfrak{s}} \triangleq \forall_{p\in\rho}.\forall_{c\in\varsigma}.\forall_{c_l\in\varsigma}.\forall_{c_r\in\varsigma}.\mathtt{pf}\ p\ c = (c_l, c_r) \implies c_l \sqsubseteq_{\mathfrak{s}} c \ \wedge\ c_r \sqsubseteq_{\mathfrak{s}} c \ \wedge\ c_l \lozenge_{\mathfrak{s}} c_r$$

We now replace the type ς with $\mathcal{A}$, the elements of a maximal lattice, and give a new definition for $\Diamond$:

$$c_l \Diamond_{\mathfrak{s}} c_r \triangleq c_l \subseteq \neg c_r \ \wedge \ c_r \subseteq \neg c_l$$

This is the most obvious choice: two processes may be run concurrently in contexts c_l and c_r respectively if the actions in c_l are a subset of those allowed in parallel with c_r, and vice-versa.

The pre-condition now becomes:

$$\forall_{p\in\rho}.\forall_{c\in\mathcal{A}}.\forall_{c_l\in\mathcal{A}}.\forall_{c_r\in\mathcal{A}}.\mathtt{pf}\ p\ c = (c_l, c_r) \Longrightarrow c_l \subseteq c \ \wedge \ c_r \subseteq c \ \wedge \ c_l \subseteq \neg c_r \ \wedge \ c_r \subseteq \neg c_l$$

The other pre-conditions also translate easily as follows, by replacing ς with $\mathcal{A}$:

$$\begin{aligned}
\mathsf{mon} &\triangleq \forall_{p\in\rho}.\forall_{c\in\mathcal{A}}.\forall_{c_l\in\mathcal{A}}.\forall_{c_r\in\mathcal{A}}.\forall_{c'\in\mathcal{A}}.\forall_{c'_l\in\mathcal{A}}.\forall_{c'_r\in\mathcal{A}}.\\
&\quad \mathtt{pf}\ p\ c = (c_l, c_r) \Longrightarrow \mathtt{pf}\ p\ c' = (c'_l, c'_r) \Longrightarrow c \subseteq c' \Longrightarrow c_r \subseteq c'_r \ \wedge \ c_l \subseteq c'_l\\
\mathsf{lft} &\triangleq \exists_{\mathtt{lft}\in\rho}.\forall_{c\in\mathcal{A}}.\exists_{c_r\in\mathcal{A}}.\mathtt{pf}\ \mathtt{lft}\ c = (c, c_r)\\
\mathsf{sym} &\triangleq \exists_{\mathtt{sym}\in\rho\to\rho}.\forall_{p\in\rho}.\forall_{c\in\mathcal{A}}.\forall_{c_l\in\mathcal{A}}.\forall_{c_r\in\mathcal{A}}.\mathtt{pf}\ p\ c = (c_l, c_r) \Longrightarrow \mathtt{pf}\ (\mathtt{sym}\ p)\ c = (c_r, c_l)\\
\mathsf{asl} &\triangleq \exists_{\mathtt{asl}\in(\rho,\rho)\to(\rho,\rho)}.\forall_{p\in\rho}.\forall_{c\in\mathcal{A}}.\forall_{c_1\in\mathcal{A}}.\forall_{c_2\in\mathcal{A}}.\forall_{c_3\in\mathcal{A}}.\\
&\quad (\exists_{c_T\in\mathcal{A}}.\mathtt{pf}\ p_1\ c = (c_T, c_3) \ \wedge \ \mathtt{pf}\ p_2\ c_T = (c_1, c_2)) \Longrightarrow \exists_{p_1\in\rho}.\exists_{p_2\in\rho}.\\
&\quad\quad \mathtt{asl}\ (p_1, p_2) = (p'_1, p'_2) \ \wedge \ (\exists_{c_T\in\mathcal{A}}.\mathtt{pf}\ p'_1\ c = (c_1, c_T) \ \wedge \ \mathtt{pf}\ p'_2\ c_T = (c_2, c_3))
\end{aligned}$$

The central question now concerns what the type ρ should be and what mathematical expression pf should encode.

8.3.1 Context Splitter A

As the simplest useful context-splitter, consider the following:

$$\begin{aligned}
\rho &= \mathcal{A}\\
\mathtt{pf}_A\ c_p^+\ c^+ &= (c^+ \sqcap c_p^+, c^+ \sqcap \neg c_p^+)
\end{aligned}$$

One specifies how a context c is to be split by supplying a second context, c_p^+. The left-hand context becomes $c^+ \sqcap c_p^+$ and the right-hand context becomes $c^+ \sqcap \neg c_p^+$. This immediately guarantees that no child context will permit actions forbidden by the parent (both are $\subseteq c^+$) and that both left and right child context will commute with each other.

Theorem 8.3.1. *The function $\mathtt{pf}_A$, when used to split contexts, obeys* $\mathsf{PRE}_{\mathfrak{s}}$, mon, lft *and* sym, *where $\mathtt{lft} \triangleq \mathbb{1}$ and $\mathtt{sym}\, c \triangleq \neg c$.*

Proof. Treat each individual pre-condition in turn:

- $\mathsf{PRE}_{\mathfrak{s}}$: It is trivially true that $c^+ \sqcap c_p^+ \subseteq c^+$ and $c^+ \sqcap \neg c_p^+ \subseteq c^+$. To prove that the left-hand child context only allows actions which may be run in parallel with the

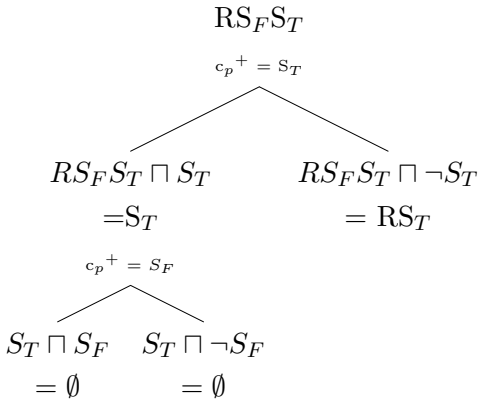
right-hand child context, we must show that $c^+ \sqcap c_p^+ \subseteq \neg(c^+ \sqcap \neg c_p^+)$, which, using the de Morgan involution, is equivalent to $c^+ \sqcap c_p^+ \subseteq \neg c^+ \sqcup c_p^+$. This is true since the left-hand side is no larger than c_p^+ and the right-hand side is no smaller than c_p^+.

The proof that $c^+ \sqcap \neg c_p^+ \subseteq \neg(c^+ \sqcap c_p^+)$ is similar.

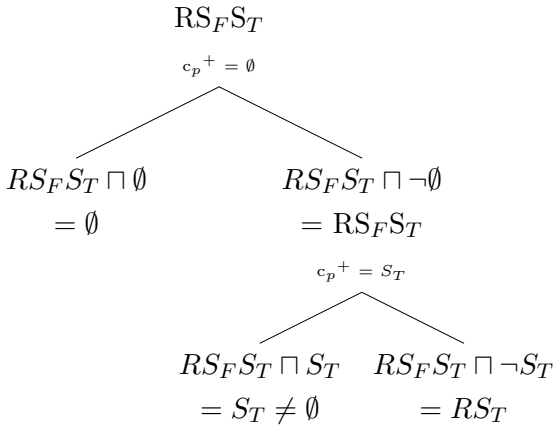
- mon: if $c^+ \subseteq c'^+$ then it is clear that both $c^+ \sqcap c_p^+ \subseteq c'^+ \sqcap c_p^+$ and $c^+ \sqcap \neg c_p^+ \subseteq c'^+ \sqcap \neg c_p^+$.
- lft: for any context c^+, $\mathtt{pf}_A\ \mathbb{1}\ c^+ = (c^+, \mathbb{0})$, since $c^+ \sqcap \neg\mathbb{1} = c^+ \sqcap \mathbb{0} = \mathbb{0}$ and $c^+ \sqcap \mathbb{1} = c^+$.
- sym: for any context c^+, if $\mathtt{pf}_A\ c_p^+\ c^+ = (c_l^+, c_r^+)$ then $\mathtt{pf}_A\ \neg c_p^+\ c^+ = (c_r^+, c_l^+)$ because $\neg\neg c_p^+ = c_p^+$.

□

This context splitter does not obey asl. Consider the RS_FS_T model. Say a program splits the $\mathbb{1}$ context RS_FS_T into $\emptyset$, $\emptyset$ and RS_T in the following manner:



There is no way that these three contexts can be obtained from RS_FS_T as a right-leaning tree. In order to guarantee that the left-most context is $\emptyset$ we must give all permissions to the right-hand side at the top level. But having done this there is no means of making one sub-context equal RS_T without making the other equal S_T, as the following diagram demonstrates:



8.3.2 Context Splitter B

In the previous example it could be noted that when attempting to split context S_T by supplying a parameter S_F it yielded two $\emptyset$ contexts even though S_T would have been possible in both. That is, the most powerful sub-context was not being inferred.

The idea behind the modified context splitter in this section is that a program running in c^+, or any sub-context of c^+, can *always* safely perform an action in context $c^+ \sqcap \neg c^+$. So Splitter A is changed to take this into account:

$$\begin{aligned} \rho &= \mathcal{A} \\ \mathtt{pf}_B\ c_p^+\ c^+ &= \mathtt{((}c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap c_p^+)\mathtt{,}(c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap \neg c_p^+)\mathtt{)} \end{aligned}$$

Theorem 8.3.2. *The function* $\mathtt{pf}_B$ *obeys* $\mathsf{PRE_s}$, lft *and* sym, *where* $\mathtt{lft} \triangleq \mathbb{1}$ *and* $\mathtt{sym}\, c \triangleq \neg c$.

Proof.

- $\mathsf{PRE_s}$: it is trivially true that $(c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap c_p^+) \subseteq c^+$ and $(c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap \neg c_p^+) \subseteq c^+$ since both left-hand sides can be no greater than c^+.

  To prove that the left-hand child context only allows actions which may be run in parallel with the right-hand child context, we must show that

  $$(c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap c_p^+) \subseteq \neg((c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap \neg c_p^+))$$

  Using the de Morgan laws, rewrite the right-hand side as follows:

  $$(c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap c_p^+) \subseteq (\neg c^+ \sqcup c^+) \sqcap (c^+ \sqcup c_p^+)$$

  From Proposition 8.2.1, the weakened distributive law, we know that $\neg c^+ \sqcup (\neg c^+ \sqcap c_p^+) \subseteq (\neg c^+ \sqcup c^+) \sqcap (\neg c^+ \sqcup c_p^+)$, so we need only prove that

  $$(c^+ \sqcap \neg c^+) \sqcup (c^+ \sqcap c_p^+) \subseteq \neg c^+ \sqcup (c^+ \sqcap c_p^+)$$

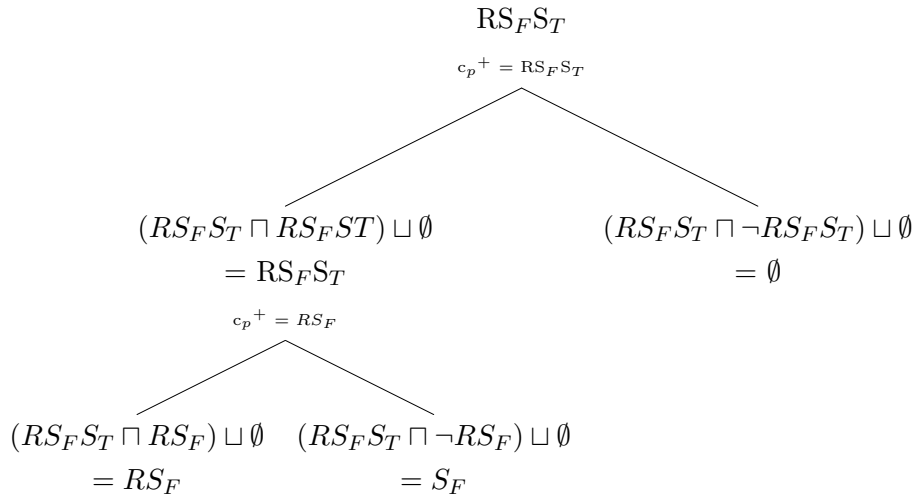
  This is straightforward, and is effectively a proof that $(a \sqcap b) \sqcup c \subseteq b \sqcup c$. The proof for the other side is similar.

- lft: for any context c^+, $\mathtt{pf}_B\ \mathbb{1}\ c^+ = \mathtt{(}c^+\mathtt{,}c^+ \sqcup \neg c^+\mathtt{)}$. The right-hand side is not given the minimum number of permissions. Instead it permits only those actions in c^+ that also commute with all actions in c^+.

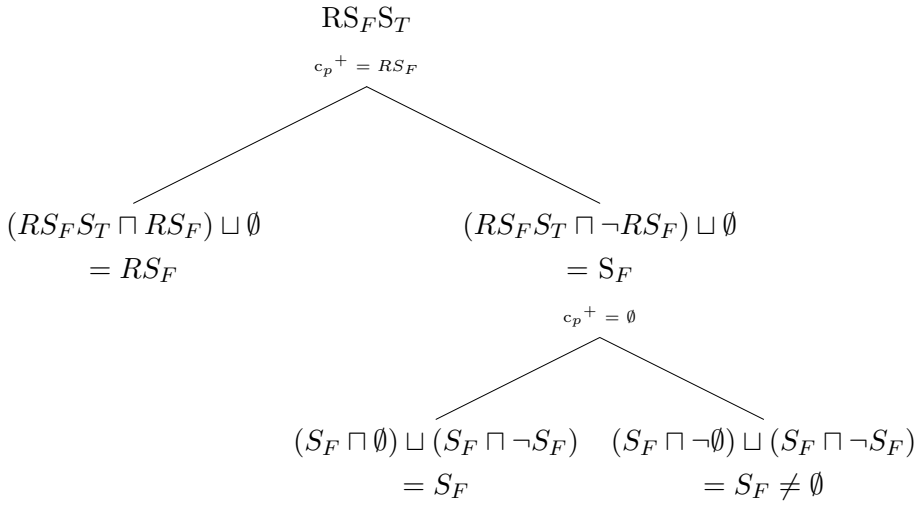
- sym: for any context c^+, if $\mathtt{pf}_B\ c_p^+\ c^+ = \mathtt{(}c_l^+\mathtt{,}c_r^+\mathtt{)}$ then $\mathtt{pf}_B\ \neg c_p^+\ c^+ = \mathtt{(}c_r^+\mathtt{,}c_l^+\mathtt{)}$ once again because $\neg\neg c_p^+ = c_p^+$.

□

This context splitter does not obey asl either, however. Once again consider the RS_FS_T model. Say a program splits the $\mathbb{1}$ context RS_FS_T into RS_F, S_F and $\emptyset$ in the following manner:



When one tries to transform this into a right-leaning tree problems are encountered. We must make the immediate left-hand context RS_F so therefore the right-hand side must become S_F. At this point, since $S_F \sqcap \neg S_F = S_F$, we are unable to restrict a sub-context to be $\emptyset$:



This second context splitter does not obey mon either. A quick counter-example is that although both

$$\begin{aligned} \mathtt{pf}_B\ (RS_FS_T)\ (RS_FS_T) &= (RS_FS_T, \emptyset) \\ \mathtt{pf}_B\ (RS_FS_T)\ (RS_F) &= (RS_F, S_F) \end{aligned}$$

and $RS_F \subseteq RS_FS_T$ it is not true that $S_F \subseteq \emptyset$.

8.3.3 Discussion

We have demonstrated two high-level mechanisms for splitting I/O contexts but neither solve the associativity problem. The example from Splitter B indicates more clearly what the fundamental difficulty is. If we split the context RS_FS_T into $(RS_FS_T, \emptyset)$ then the $\emptyset$ is necessary. But if RS_FS_T is further split into (RS_F, S_F) then immediately the original $\emptyset$ becomes too weak since it could be replaced with S_F. Therefore it seems that the notion of "most general sub-context" is an inherently global one and not suited to this sort of partitioning.

It should be noted that this problem cannot be solved by somehow building a larger structure which contains the contexts missing from the maximal lattice. This is because, in the example, RS_FS_T's maximal lattice *isn't* missing any contexts – the fact that a maximal lattice is a sub-lattice of the original Boolean algebra has nothing whatsoever to do with this. In the same way, the non-distributivity of maximal lattices is a separate, unrelated problem because model RS_FS_T is distributive.

It is hard to see how this associativity problem could be solved without significant modifications (perhaps there is a simple, elegant solution but I cannot guess what it might be). There are three possible directions:

- Develop a higher-order notion of a context. This would mirror the approach taken in Chapter 7. A context would then become one of two things – either a "real" context or a function which computes a real context given some other information.

- Tighten the properties which maximal lattices obey, thereby excluding certain I/O models which cause the above problem. It is not obvious how much this would restrict the flexibility our our models, however.

- Use an n-way `par` operation and an n-way context splitter, instead of binary ones.

8.4 Future directions and related work

This chapter describes a mathematical structure which we believe is sufficiently general to allow one to model the essence of any system of I/O contexts, including those shown previously in this dissertation. Through this generality one can describe the aspects of I/O contexts which are most critical while ignoring implementation details that are of little importance.

There is quite a lot of evidence to suggest that maximal lattices are the correct abstraction for this task. That we can give entirely general algorithms for splitting any context, allowing us to unify notions such as "all permissions" and "minimum permissions", is reason enough to believe that maximal lattices have some merit. And, importantly, as argued above, the associativity problem is not really related to this choice of structure.

Ideally, maximal lattices should be embedded directly within the language semantics as a replacement for the functions `ap` and `pf`. So this chapter really just constitutes an extended future-work section. The current language semantics does have its benefits. Most notably, the confluence proof was machine-verified and it is unclear how maximal lattices could be embedded into a domain theoretic LCF-style theorem prover.

8.4.1 Similarities to other algebras

The author was unable to find a previously studied lattice or algebra that obeys the exact properties of maximal lattices as outlined in this chapter. One reason for this may be that effectively all mathematically interesting lattices are distributive.

If a maximal lattice *is* distributive then it forms a quasi-Boolean algebra (or de Morgan algebra). In a quasi-Boolean algebra $\sqcup$, $\sqcap$ and $\neg$ obey all the properties of a Boolean algebra with the exception of the laws of the excluded middle (so it is not true that $c^+ \sqcup \neg c^+ = \mathbb{1}$ or $c^+ \sqcap \neg c^+ = \mathbb{0}$). Quasi-Boolean algebras were fully classified in [11], and a description of their properties may be found in [100]. These algebras have been used to model many-valued extensions to classical logic.

Our original Boolean Algebra contained the complement operator $\sim$. This operator itself cannot be used directly since it may not preserve membership of the sub-lattice. Some structures, however, such as Łukasiewicz n-valued algebras [13], solve this by instead using a finite family of related complement operators which are slightly weaker.

The author investigated this, defining two unary operators $\triangle$ and $\triangledown$ as follows:

$$\triangledown c \stackrel{\triangle}{=} \neg\neg\sim c \qquad\qquad \triangle c \stackrel{\triangle}{=} \neg\sim\neg c$$

These gave rise to some intriguing results:

Theorem 8.4.1. *The operators $\triangle$ and $\triangledown$ obey the following properties:*

$$\begin{array}{lrcl}
\textit{(i)} & \triangledown\neg c^+ & = & \neg\triangle c^+ \\
\textit{(ii)} & \triangledown(c_1^+ \sqcap c_2^+) & = & \triangledown c_1^+ \sqcup \triangledown c_2^+ \\
\textit{(iii)} & \triangle(c_1^+ \sqcup c_2^+) & = & \triangle c_1^+ \sqcap \triangle c_2^+ \\
\textit{(iv)} & c^+ \sqcup \triangledown c^+ & = & \mathbb{1} \\
\textit{(v)} & c^+ \sqcap \triangle c^+ & = & \mathbb{0} \\
\textit{(vi)} & \triangle c^+ & \subseteq & \triangledown c^+ \\
\textit{(vii)} & \triangle\triangle\triangle c^+ & = & \triangle c^+ \\
\textit{(viii)} & \triangledown\triangledown\triangledown c^+ & = & \triangledown c^+
\end{array}$$

$$\begin{array}{lrcccl}
\textit{(ix)} & \triangledown\triangledown c^+ & \subseteq & c^+ & \subseteq & \triangle\triangle c^+ \\
\textit{(x)} & \triangle\mathbb{1} & = & \mathbb{0} & = & \triangledown\mathbb{1} \\
\textit{(xi)} & \triangle\mathbb{0} & = & \mathbb{1} & = & \triangledown\mathbb{0}
\end{array}$$

Proof. See Appendix B.1. □

These extra operators of maximal lattices sadly have little in common with Łukasiewicz algebras. One main problem is that $\triangle$ and $\triangledown$ do not both distribute over $\sqcup$ and $\sqcap$. The proofs of the above properties are included in this dissertation mainly for curiosity value.

Another well-known weakening of a Boolean algebra is a Heyting algebra [69], primarily famous as an algebraic model of intuitionistic logic. In a Heyting algebra it is not true that $x \sqcup \neg x = \mathbb{1}$ but, unlike with a de Morgan algebra, $x \sqcap \neg x = \mathbb{0}$ holds, and this is usually not true of maximal lattices. Heyting algebras are also always distributive.

Despite being fundamentally different to maximal lattices, Heyting algebras have a surprising similarity in a smaller way. In Heyting algebras, $\sim\!\sim\!\sim x = \sim x$, $x \subseteq \sim\!\sim x$ and $x \subseteq y \implies \sim y \subseteq \sim x$. The $\neg$ operator obeys these very properties on our original Boolean algebra before we remove elements to form a maximal lattice, and $\triangle$ obeys these properties for a maximal lattice.

Chapter 9

Conclusions and future work

9.1 Conclusions

We have presented a detailed description of the Curio language. This language handles the semantics of I/O and concurrency by modelling the API explicitly. It was shown how one can model the Haskell 98 I/O interface using this approach and how Curio preserves many of the existing semantic properties which functional languages enjoy.

It is unusual for a language's semantics for I/O to include an entirely general-purpose model of the API. Since the over-sequencing problem with monadic I/O is so dependent on the specifics of the actual API this seems to be a legitimate and logical starting point – yet it is a solution which we have not seen. This dissertation sketches how this may be achieved in a rudimentary yet powerful fashion.

Another quite unusual feature is how we use state-transformers to model concurrency and communication. These state-transformers are somewhat unusual since it is still really just individual actions that are modelled with actual state-transformers, not whole programs. But it does give rise to interesting notions of program equivalence in Chapter 7.

We advocate the use of proof tools, where possible, to verify theoretical results. The Sparkle proof-assistant was particularly suited to proving properties about Curio because almost all of the features of Curio are inherited directly from the metalanguage.

9.2 Future work

Curio is still largely at the proof-of-concept stage. It does not yet have an implementation which lets the user write real programs and, for this reason, we do not have any significant user experience at writing such programs. The metalanguage encoding exists, of course, but the types are restricted and any actual I/O will only modify the semantic model. Similarly, we have not used it to do large, I/O-intensive proofs.

There are some specific areas of future research which present themselves to us.

9.2.1 More exotic types

The most immediate limitation to Curio in its current state is its mundane type system. We made no attempt in this dissertation to modify the traditional Hindley-Milner system in any sense. This affects how we specify I/O models and also how we write programs.

Firstly, the five types which parameterise an I/O model are monomorphic. One example of how this leads to limitations it that although communication channels may be allocated dynamically, we cannot create one that can send values of an arbitrary type. In the real world API in Chapter 6, channels only allow characters to be sent and received. This does not sound particularly problematic but it is not immediately clear how it could be solved. Other aspects of the I/O models are also a little clumsy, such as our rigidly defined types ν and α for return values and actions respectively. These are inconvenient to use and quickly become cluttered in the presence of combinators.

Secondly, it would be interesting to see if some or all of the runtime checks required by Curio could be performed statically. I/O contexts are quite similar to types, since a program's outermost I/O context does not change as the program reduces over time. Ideally this would lead to the complete removal of the `test` command – any attempt to perform an action not permitted by a program's context would be detected at compile time. In reality, this would not be easy. Types would most likely become heavily annotated, and type inference would almost certainly be undecidable for anything but the simplest of cases. It is, nonetheless, a compelling end goal.

Existing approaches do also require some runtime checks. In Clean, the type `*World` denotes the rest of the world, in which some (unspecified) files are accessible – one must test at runtime to see if a particular file can be accessed. Something similar occurs also in Brisk.

9.2.2 GUI systems

We did not make any attempt to model graphical user interfaces in Curio. This is almost certainly the most important aspect of real I/O which we ignored.

GUI applications present ample opportunities for concurrency. Often one thinks of an individual on-screen window as a separate process which waits for input such as mouse clicks, modifies its own piece of screen real estate and perhaps its own state, and then communicates with other processes. It would be very interesting to see if the low-level GUI API could be encoded elegantly, and if existing GUI libraries such as Fudgets, Yampa and Object I/O could be expressed better in such a system.

9.2.3 A simpler, core calculus

The design of Curio has been influenced greatly by our desire to verify properties using a proof-assistant, an LCF style one in particular. The I/O models presented in Chapter 2 are

unnecessarily infected by this, and this leads to a messier confluence proof (when $\mathtt{wa}\ a\ w = \bot$, for example) and the unintuitive failure of certain equivalence results (as mentioned at the end of Chapter 7).

We pride ourselves in being able to machine-verify our results, but perhaps a simpler calculus would be in order to explore the more formal capabilities of Curio. Other logical frameworks could be used instead to machine-verify results, and these would probably force us to explicitly give the semantics of a small PCF-like language. In doing so we would then be able to give a more complete axiomatic semantics, and formally prove contextual equivalence results. This would, however, be at the expense of only having a tiny language to work with.

As well as a minimal language, a minimal notation for defining new I/O models would also be desirable. It is a shame that so much boilerplate is needed to define even tiny I/O models such as $\mathfrak{istr}$. For a start, the maximal lattice structure shown in Chapter 8 appears to be general enough for it to be used almost immediately. This would automatically give programs better equational properties and make the development of combinators much more straightforward (recall how we needed the types Cxt and Splitter in Chapter 5).

9.2.4 Exceptions

Other possible future work would be the inclusion of exceptions in Curio. Concurrent Haskell has exceptions [88, 91], but these are non-deterministic within the `IO` monad. It would be interesting to see if exceptions could be added to Curio in a way that did not introduce non-determinism, even in the presence of concurrency.

Exceptions could also be used as a substitute for the runtime failure required to guarantee confluence in Curio. One example of this is the way an action fails outright if it is not permitted by its context. Another example is the I/O model $\mathfrak{istr}$, which obeys $\mathsf{PRE}_{\mathfrak{s}}$ only because writing to an already full I-structure causes failure. In practice, it would be more satisfactory if instead an exception was thrown. The most immediate difficulty this raises, however, is what the contents of the world state should be once the exception is caught.

9.2.5 Other ideas

Our I/O models are designed to be intuitive in the sense that the world state looks something like we expect it to be – a file is a list of characters and some associated information about its current readers and writers. But this requires a precondition to guarantee confluence. I/O models which are entirely deterministic *by construction* could be an interesting future direction. The combinators of Chapter 5 effectively took the first step.

It is possible that the co-inductive semantics for program equality has useful parallels in the theory of process calculi such as CCS. As well as theoretical relationships, we could also model terminal I/O in the same "observable" style as that used by Haskell. Terminal I/O in model $\mathfrak{term}$ is described using a datatype which gives the semantics of the user. We

could equally well assume the existence of an entirely unspecified "terminal user process" which communicates with the user via communication buffers. One could then reason about programs co-inductively by showing that they can be observed to react identically to user responses.

The I/O interface used by Clean and Haskell is different on account of Clean's unique types. Curio could be used as a basis for unifying the interface, and give a common semantics to I/O for both languages. Curio could also inspire other approaches to I/O in existing languages – possibly even as the basis for formal correctness proofs. Clean, for example, allows the order in which actions are performed to be loosened but it does not permit communication. Perhaps some primitives which behave in a similar way to `newChannel`, `putChar` and `getChar` used in Chapter 6 could alleviate this if implemented correctly.

Curio was intended to help to give a semantics to deterministic concurrency. It is possible that it could also be useful in attempting to understand and prove properties about traditional, non-deterministic concurrency.

Another future direction could be an attempt to give a more abstract semantics to Curio. Denotational semantics and domain theory sometimes take place within the more general framework of category theory. When this occurs, as described, for example, by Gibbons in [35], a category is a language, an object of that category is a type/domain, an arrow in that category is a program, and a functor is a type constructor. We have not touched upon this at all in this dissertation, but category theory could possibly provide elegant generalisations for our work.

Appendix A

Implementation details

Sections A.1, A.2 and A.3 correspond to the implementation details for Chapters 3 & 4, Chapter 5 and Chapter 6 respectively. This appendix only contains the "left-overs" omitted from preceding chapters – we do not repeat any definitions given in the main document.

A.1 Curio implementation

The implementations are slightly tidied versions of the actual ones. Apart from naming conventions, and some omitted strictness annotation, the most substantial change is how values of type Guess/Route are used. In the real implementation these are both identical and implemented as a strict list of Bool. If the list is being used as a Guess then we have a mechanism which pads this list so that it behaves as if it were infinite.

A.1.1 Reduction in Curio

Figure A.1 contains the functions `next` and `rdce` which along with some helper functions is the complete encoding for single-step reduction in Curio.

A.1.2 Re-implementing $\texttt{next}_\varsigma$

```
advS :: IOModel ν α ρ ω ς → ς → Route → Prog ν α ρ → Prog ν α ρ
advS s            c []    (Bind (Ret v) f) = f v
advS s            c r     (Bind m f)       = Bind (advS s c r m) f
advS (_,_,ap,_)   c []    (Test a mt mf)   =
  if (ap c a) then mt else mf
advS (_,_,_,pf)   c []    (Par p (Ret vl) (Ret vr) vf) = Ret (vf vl vr)
advS s@(_,_,_,pf) c (L:r) (Par p ml mr vf) =
  Par p (advS s (fst (pf p c)) r ml) mr vf
advS s@(_,_,_,pf) c (R:r) (Par p ml mr vf) =
  Par p ml (advS s (snd (pf p c)) r mr) vf
```

```
next :: IOModel ν α ρ ω ς → Guess → ς → (ω,Prog ν α ρ) → Reduction (ω,Prog ν α ρ)
next s            g c (w, Ret v)          = Converged
next s            g c (w, Bind (Ret v) f) = Reduct (w, f v)
next s            g c (w, Bind m f)       = case (next s g c (w,m)) of
  Converged      -> Converged
  Reduct (w1,m1) -> Reduct (w1, Bind m1 f)
next (af,wa,ap,_) g c (w, Action a) | ap c a =
  if (wa a w) then Converged
              else (case (af a w) of
                      (w1,v1) -> Reduct (w1, Ret v1))
next (_,_,ap,_)   g c (w, Test a mt mf)   =
  Reduct (w, if (ap c a) then mt else mf)
next s@(_,_,_,pf) g c (w, Par p ml mr vf) = case (pf p c) of
  (cl,cr) -> nextPar s cl cr ml mr p vf w g

nextPar :: IOModel ν α ρ ω ς → ς → ς → Prog ν α ρ → Prog ν α ρ → ρ →
  (ν → ν → ν) → ω → Guess → Reduction (ω,Prog ν α ρ)
nextPar s cl cr ml mr p vf w (d:g) = case ml of
  Ret vl -> case mr of
    Ret vr -> Reduct (w, Ret (vf vl vr))
    _      -> doR
  _      -> case mr of
    Ret vr -> doL
    _      -> if (d==R) then (fstReduction doR doL)
                        else (fstReduction doL doR)
 where doL = case (next s g cl (w,ml)) of
         Reduct (w1,ml1) -> Reduct (w1, Par p ml1 mr vf)
         Converged       -> Converged
       doR = case (next s g cr (w,mr)) of
         Reduct (w1,mr1) -> Reduct (w1, Par p ml mr1 vf)
         Converged       -> Converged

fstReduction :: Reduction β → Reduction β → Reduction β
fstReduction (Reduct x) _  = (Reduct x)
fstReduction Converged  r  = r

nextWrap :: IOModel ν α ρ ω ς → ς → (ω,Prog ν α ρ,[Guess],Bool) →
  (ω,Prog ν α ρ,[Guess],Bool)
nextWrap s c (w,m,(g:gs),False) = case (next s c g (w,m)) of
  Converged      -> (w,m,gs,True)
  Reduct (w1,m1) -> (w1,m1,gs,False)

rdce :: IOModel ν α ρ ω ς → Int → [Guess] → ς → (ω,Prog ν α ρ) → (ω,Prog ν α ρ)
rdce s i gs c (w,m) | i>=0 =
  case (iterate (nextWrap s c) (w,m,gs,False) !! i) of
    (w1,m1,_,False) -> (w1,m1)
```

Figure A.1: Implementation of $\texttt{next}_5$ and $\texttt{rdce}_5$

```
advA :: ς → Route → Prog ν α ρ → Prog ν α ρ
advA v r     (Bind m f)       = Bind (advA v r m) f
advA v []    (Action a)       = Ret v
advA v (L:r) (Par p ml mr vf) = Par p (advA v r ml) mr vf
advA v (R:r) (Par p ml mr vf) = Par p ml (advA v r mr) vf


nextR :: IOModel ν α ρ ω ς → Guess → ς → (ω, Prog ν α ρ) → Redex (Route, RxType α)
nextR s           g c (Ret v)          = NoRedex
nextR s           g c (Bind (Ret v) f) = Redex ([], Silent)
nextR s           g c (Bind m f)       = nextR s c g m
nextR (_,wa,ap,_) g c (Action a) | ap c a =
  if (wa a w) then NoRedex else (Redex ([],Action a))
nextR s           g c (Test a mt mf)   = Redex ([], Silent)
nextR (_,_,_,pf)  g c (Par p ml mr vf) = case (pf p c) of
  (cl,cr) -> nextRPar s cl cr g w ml mr

nextRPar s cl cr (d:g) w ml mr = case ml of
  Ret vl -> case mr of
    Ret vr -> Redex ([],Silent)
    _      -> doR
  _      -> case mr of
    Ret vr -> doL
    _      -> if (d==R) then (fstRedex doR doL)
                        else (fstRedex doL doR)
 where
   doL = case (nextR s g cl w ml) of
     NoRedex     -> NoRedex
     Redex (r,x) -> Redex ((L:r), x)
   doR = case (nextR s g cr w mr) of
     NoRedex     -> NoRedex
     Redex (r,x) -> Redex ((R:r), x)
```

A.1.3 Re-implementing $\texttt{nextR}_s$

```
preorder :: Tree β → [β]
preorder (Leaf x)       = [x]
preorder (Branch t1 t2) = preorder t1 ++ preorder t2
```

```
shuffle :: Guess → Tree β → Tree β
shuffle g     (Leaf x)       = Leaf x
shuffle (L:g) (Branch t1 t2) = Branch (shuffle g t1) (shuffle g t2)
shuffle (R:g) (Branch t1 t2) = Branch (shuffle g t2) (shuffle g t1)

firstRx :: Redex β → Redex β → Redex β
firstRx (Redex x) _  = Redex x
firstRx NoRedex   r  = r

check :: IOModel ν α ρ ω ς → ω → Redex (Route,(ς,RxType α)) →
  Redex (Route,RxType α)
check s w NoRedex = NoRedex
check s w (Redex (r, (c,Silent)))   = Redex (r,Silent)
check s w (Redex (r, (c,Action a))) | ap c a =
  if (wa a w) (Redex (r,Silent)) (Redex (r,Action a))

addDir :: Dir → Redex (Route,β) → Redex (Route,β)
addDir d NoRedex       = NoRedex
addDir d (Redex (r,x)) = Redex ((d:r),x)

redexTree :: IOModel ν α ρ ω ς → ς → Prog ν α ρ →
  Tree (Redex (Route,(ς,RxType α)))
redexTree s c (Ret v)                    = Leaf NoRedex
redexTree s c (Bind (Ret v) f)           = Leaf (Redex ([],(c,Silent)))
redexTree s c (Bind m f)                 = redexTree s c m
redexTree s c (Action a)                 = Leaf (Redex ([],(c,Action a)))
redexTree s c (Test a mt mf)             = Leaf (Redex ([],(c,Silent)))
redexTree s c (Par p (Ret vl) (Ret vr) vf) = Leaf (Redex ([],(c,Silent)))
redexTree s@(_,_,_,pf) c (Par p ml mr vf)  = case (pf p c) of
  (cl,cr) -> Branch (mapTree (addDir L) (redexTree s cl ml))
                    (mapTree (addDir R) (redexTree s cr mr))

mapTree :: (β → γ) → Tree β → Tree γ
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Branch t1 t2) = Branch (mapTree f t1) (mapTree f t2)
```

A.2 Combinators

A.2.1 Map functions

```
lkpM :: ∀β. ∀γ. Eq γ ⇒ γ → (γ → β) → β
lkpM s m | s==s = m s

ovwM :: ∀β. ∀γ. Eq γ ⇒ γ → β → (γ → β) → (γ → β)
ovwM s v m = \s1 -> if (s==s1) then v else (lkpM s1 m)

maskM :: ∀β. ∀γ. Eq γ ⇒ (γ → Bool) → β → (γ → β) → (γ → β)
maskM f v m = \s -> if (f s) then v else (lkpM s m)

stM :: ∀β. ∀γ. ∀δ. Eq γ ⇒ (β → (β,δ)) → γ → (γ → β) → (γ → β,δ)
stM f s = \m -> case (f $! lkpM s m) of (v,x) -> (ovwM s v1 m, x)

ovwL :: ∀β. Int → β → [β] → [β]
ovwL 0 b (b1:bs) = (b:bs)
ovwL 0 b []      = undef
ovwL i b (b1:bs) | i>0 = b1 : (ovwL (i-1) b bs)

newL :: ∀β. Int → β → [β] → [β]
newL i b bs | i==length bs = bs ++ [b]

stL :: ∀β. ∀δ. (β → (β,δ)) → Int → [β] → ([β],δ)
stL f i bs = case (f $! (bs!!i)) of (b1,x) -> (ovwL i b1 bs, x)
```

A.2.2 Pool functions

```
nextP :: ∀β. Loc → Pool β → Int
nextP _     PoolLeaf = 0
nextP []    (PoolNode p1 xs p2) = length xs
nextP (L:l) (PoolNode p1 xs p2) = nextP l p1
nextP (R:l) (PoolNode p1 xs p2) = nextP l p2

lkpP :: ∀β. HndP → Pool β → β
lkpP ([]   ,i) (PoolNode p1 xs p2) = xs !! i
lkpP ((L:l),i) (PoolNode p1 xs p2) = lkpP (l,i) p1
lkpP ((R:l),i) (PoolNode p1 xs p2) = lkpP (l,i) p2

ovwP :: ∀β. HndP → β → Pool β → Pool β
ovwP ([]   ,i) x (PoolNode p1 xs p2) = PoolNode p1 (ovwL i x xs) p2
ovwP ((L:l),i) x (PoolNode p1 xs p2) = PoolNode (ovwP (l,i) x p1) xs p2
ovwP ((R:l),i) x (PoolNode p1 xs p2) = PoolNode p1 xs (ovwP (l,i) x p2)
```

```
newP :: ∀β. HndP → β → Pool β → Pool β
newP h@(l,i) x p = newP' h x (padP l p)

newP' :: ∀β. HndP → β → Pool β → Pool β
newP' ([]   ,i) x (PoolNode p1 xs p2) = PoolNode p1 (newL i x xs) p2
newP' ((L:l),i) x (PoolNode p1 xs p2) = PoolNode (newP' (l,i) x p1) xs p2
newP' ((R:l),i) x (PoolNode p1 xs p2) = PoolNode p1 xs (newP' (l,i) x p2)

padP :: ∀β. Loc → Pool β → Pool β
padP l     PoolLeaf            = padP l (PoolNode PoolLeaf [] PoolLeaf)
padP []    (PoolNode p1 xs p2) = PoolNode p1 xs p2
padP (L:l) (PoolNode p1 xs p2) = PoolNode (padP l p1) xs p2
padP (R:l) (PoolNode p1 xs p2) = PoolNode p1 xs (padP l p2)
```

A.2.3 Helper functions for **Cxt** ς

```
apCxt :: ∀ς. ∀α. (ς → α → Bool) → Cxt ς → α → Bool
apCxt ap CxtB          a = False
apCxt ap (Cxt c False) a = ap c a
apCxt ap (Cxt c True)  a = True

pfCxt :: ∀ς. ∀ρ. (ρ → ς → (ς,ς)) → Splitter ρ → Cxt ς → (Cxt ς,Cxt ς)
pfCxt pf p         CxtB      = (CxtB , CxtB)
pfCxt pf AllLeft   c         = (c    , CxtB)
pfCxt pf AllRight  c         = (CxtB , c   )
pfCxt pf (Split p) (Cxt c b) = case (pf p c) of
  (cl,cr) ->                   (Cxt cl False, Cxt cr False)

splitM :: ∀ρ. ∀ς. ∀γ. Eq γ ⇒ (ρ → ς → (ς,ς)) → (γ → Bool) → [(γ,Splitter ρ)] →
  (γ → Cxt ς) → (γ → Cxt ς,γ → Cxt ς)
splitM pf im [] cm = (maskM (not . im) CxtB cm, maskM im CxtB cm)
splitM pf im ((s,p):ps) cm = case (splitM pf im ps cm) of
  (cml,cmr) -> case (pfCxt pf p (lkpM s cm)) of
    (cl,cr) -> (ovwM s cl cml, ovwM s cr cmr)
```

A.2.4 Determining a process' location

The following program computes its current location by continually using `test` in conjunction with PROBE. `locList` is an infinite list of locations.

```
probeLoc :: Prog_LMtoM_s Loc
probeLoc = probeFromList locList
  where
    probeFromList (l:ls) = test (l,Probe) (return l) (probeFromList ls)

locList :: [Loc]
locList = concat $ map locListInt [0..]
  where locListInt 0 = [[]]
        locListInt i =
          let list1 = locListInt (i-1)
          in  map (L :) list1 ++ map (R :) list1
```

A.3 Real world I/O semantics

A.3.1 File action semantics

```
allStale :: Pool FPtr → Bool
allStale p = foldlP (\c ptr -> case ptr of
                         FPtr i -> c+1
                         Stale  -> c) 0 p

foldlP :: ∀β. ∀γ. (β → γ → β) → β → Pool γ → β
foldlP f x PoolLeaf = x
foldlP f x (PoolNode pl ys pr) = foldlP f (foldl f (foldlP f x pl) ys) pr

doWFA :: WholeFileAction → ω_File → (ω_File,ν_File)
doWFA FRemove       _ = (NoFile,   RNull)
doWFA (FPutChar c) (File cs (WrPtr i m))
  | i>=0 && i<=length cs && (m/=AppendMode || i<length cs)
    = (File (ovwL i c cs) (wrPtr (i+1) m),  RNull)
doWFA (HWrOpen m)  (File cs (RdPtrs p)) | allStale p = case m of
  AppendMode -> (File cs (WrPtr (length cs) m), RNull)
  WriteMode  -> (File [] (WrPtr 0           m), RNull)
  _          -> (File cs (WrPtr 0           m), RNull)
doWFA FWrIsOpen (File cs ptr) = (File cs ptr, RBool (case ptr of
  RdPtrs p  -> False
  WrPtr i m -> True))
doWFA FIsOpen   (File cs ptr) = (File cs ptr, RBool (case ptr of
  RdPtrs p  -> not (allStale p)
  WrPtr i m -> True))
```

```
doFPA :: FilePtrAction → [Char] → Int → (FPtr,ν_File)
doFPA FGetChar        cs i | i>=0 && i<length cs
  = (FPtr (i+1), RChar (cs !! i))
doFPA FIsEOF          cs i
  = (FPtr i    , RBool (i==length cs))
doFPA (FSeek smode j) cs i = (FPtr i1, RNull)
  where i1 = case smode of
    AbsoluteSeek -> j
    RelativeSeek -> i+j
    SeekFromEnd  -> length cs - 1 - j
doFPA FLookAhead      cs i | i>=0 && i<length cs
  = (FPtr i, RChar (cs !! i))
doFPA FClosePtr       cs i = (Stale, RNull)
doFPA FileSize        cs i = (FPtr i, RInt (length cs))
doFPA FGetPosn        cs i = (FPtr i, RInt i)
```

A.3.2 I/O library semantics

```
hFileSize :: Handle → Prog_io Int
hFileSize (FileHnd n p) = do
  Left (Right (RInt i)) <-
    actionL (Left (Right (n, FilePtrAct p FileSize)))
  return i
hFileSize _             = return 0

HandlePosn = (Int,Handle)

hGetPosn :: Handle → Prog_io HandlePosn
hGetPosn (FileHnd n p) = do
  Left (Right (RInt i))
    <- actionL (Left (Right (n,FilePtrAct p FGetPosn)))
  return (i, FileHnd n p)

hSetPosn :: HandlePosn → Prog_io ()
hSetPosn (i, h) = hSeek h AbsoluteSeek i

hSeek :: Handle → SeekMode → Int → Prog_io ()
hSeek (FileHnd n p) m i = do
  actionL (Left (Right (fn, FilePtrAct p (FSeek m i))))
  return ()
```

```
hWaitForInput :: Handle → Prog_io ()
hWaitForInput h = hLookAhead h >>= return ()

hLookAhead :: Handle → Prog_io Char
hLookAhead StdInHnd = do
  Left (Left c) <- actionL (Left (Left GetC))
  return c
hLookAhead (FileHnd n p) =
  actionL (Left (Right (n,FilePtrAct p FLookAhead))) >>=
    \(Left (Right (RChar c))) -> return c
hLookAhead (ChanRdHnd h)   =
  actionL (Right (Left (DynAct h ChLook))) >>=
    \(Right (Left (Left (Just c)))) -> return c
hLookAhead (QSemRdHnd h) = do
  actionL (Right (Right (DynAct h SWait)))
  actionL (Right (Right (DynAct h SSignal)))
  return 'X'

hIsClosed :: Handle → Prog_io Bool
hIsClosed h = hIsOpen h >>= \b -> return (not b)

newQSem :: Prog_io (Handle,Handle)
newQSem = do
  Right (Right (Right h)) <- actionL (Right (Right Next))
  actionL (Right (Right (Alloc h)))
  return (QSemRdHnd h, QSemWrHnd h)

hIsReadable :: Handle → Prog_io Bool
hIsReadable h = case h of
  StdOutHnd          -> return False
  ChanWrHnd h        -> return False
  QSemWrHnd h        -> return False
  _                  -> return True

doesFileExist :: FilePath → Prog_io Bool
doesFileExist n = do
  Left (Right (RBool b)) <- actionL (Left (Right (n,FDoesExist)))
  return b
```

```
fIsOpen :: FilePath → Prog_io Bool
fIsOpen n = do
  Right (Left (RBool b)) <-
    actionL (Right (Left (n,WholeFileAction FIsOpen)))
  return b

hIsSeekable :: Handle → Prog_io Bool
hIsSeekable (FileHnd n p) = return True
hIsSeekable _             = return False

fAllowedW :: FilePath → Prog_io Bool
fAllowedW n = hAllowed (FileHnd n WPtr)

fAllowedR :: FilePath → Prog_io Bool
fAllowedR n = testL (Left (Right (n,FDoesExist)))
```

A.3.3 Implementation details for `parIO`

```
lkpLM :: Eq γ ⇒ γ → [(γ,β)] → Maybe β
lkpLM c [] = Nothing
lkpLM c ((c1,b):xs) | c==c1     = Just b
                    | otherwise = lkpLM c xs

ovwLM :: Eq γ ⇒ γ → β → [(γ,β)] → [(γ,β)]
ovwLM c b []                        = [(c,b)]
ovwLM c b ((c1,b1):xs) | c==c1      = (c,b):xs
                       | otherwise = (c1,b1):(ovwLM c b xs)

splitHndsTerm :: [Handle] → [Handle] → ρ_Term
splitHndsTerm hsl hsr =
  foldr splitHndTermL (foldr splitHndTermR (TCxt False False) hsr) hsl

splitHndsTermL, splitHndsTermR :: Handle → ρ_Term → ρ_Term
splitHndTermL StdInHnd  (TCxt bp bg) = (TCxt bp True)
splitHndTermL StdOutHnd (TCxt bp bg) = (TCxt True bg)
splitHndTermL _         p            = p
splitHndTermR StdInHnd  (TCxt bp bg) = (TCxt bp False)
splitHndTermR StdOutHnd (TCxt bp bg) = (TCxt False bg)
splitHndTermR _         p            = p
```

```
splitHndsFiles :: [Handle] → [Handle] → [(String,Splitter ρ_File)]
splitHndsFiles hsl hsr =
  foldr splitHndFileL (foldr splitHndFileR [] hsr) hsl


splitHndFileL :: [Handle] → [(String,Splitter ρ_File)] → [(String,Splitter ρ_File)]
splitHndFileL (FileHnd n WPtr)     nmap = ovwLM n AllLeft nmap
splitHndFileL (FileHnd n (RPtr h)) nmap = case (lkpLM n nmap) of
  Just (Split hs) -> ovwLM n (Split ((h,L):hs)) nmap
  _               -> ovwLM n (Split [(h,L)]) nmap
splitHndFileL _                    nmap = nmap


splitHndFileR :: [Handle] → [(String,Splitter ρ_File)] → [(String,Splitter ρ_File)]
splitHndFileR (FileHnd n WPtr)     nmap = ovwLM n AllRight nmap
splitHndFileR (FileHnd n (RPtr h)) nmap = case (lkpLM n nmap) of
  Just (Split hs) -> ovwLM n (Split ((h,R):hs)) nmap
  _               -> ovwLM n (Split [(h,R)]) nmap
splitHndFileR _                    nmap = nmap


splitHndsChans :: [Handle] → [Handle] → [(HndP,Splitter ρ_Chan)]
splitHndsChans hsl hsr = map
  (\(h,(mbs,mbr)) -> (h, Split (ChCxt (fromJustB mbs) (fromJustB mbr))))
  (foldr splitHndChanL (foldr splitHndChanR [] hsr) hsl)


fromJustB :: Maybe Bool → Bool
fromJustB (Just b) = b
fromJustB Nothing  = True


splitHndChanL :: Handle →
  [(HndP,(Maybe Bool,Maybe Bool))] → [(HndP,(Maybe Bool,Maybe Bool))]
splitHndChanL (ChanRdHnd h) hmap = case (lkpLM h hmap) of
  Just (mbs, mbr) -> ovwLM h (mbs, Just False)     hmap
  _               -> ovwLM h (Nothing, Just False) hmap
splitHndChanL (ChanWrHnd h) hmap = case (lkpLM h hmap) of
  Just (mbs, mbr) -> ovwLM h (Just False, mbr)     hmap
  _               -> ovwLM h (Just False, Nothing) hmap
splitHndChanL _             hmap = hmap
```

```
splitHndChanR :: Handle →
  [(HndP, (Maybe Bool, Maybe Bool))] → [(HndP, (Maybe Bool, Maybe Bool))]
splitHndChanR (ChanRdHnd h) hmap = case (lkpLM h hmap) of
  Just (mbs, mbr) -> ovwLM h (mbs, Just True)     hmap
  _               -> ovwLM h (Nothing, Just True) hmap
splitHndChanR (ChanWrHnd h) hmap = case (lkpLM h hmap) of
  Just (mbs, mbr) -> ovwLM h (Just True, mbr)     hmap
  _               -> ovwLM h (Just True, Nothing) hmap
splitHndChanR _             hmap = hmap

splitHndsQSems :: [Handle] → [Handle] → [(HndP, Splitter ρ_QSem)]
splitHndsQSems hsl hsr =
  foldr splitHndQSemL (foldr splitHndQSemR [] hsr) hsl

splitHndsQSemL :: Handle → [(HndP, Splitter ρ_QSem)] → [(HndP, Splitter ρ_QSem)]
splitHndQSemL (QSemRdHnd h) hmap = ovwLM h (Split L) hmap
splitHndQSemL _             hmap = case (lkpLM h hmap) of
  Just x   -> hmap
  Nothing  -> ovwLM h (Split L) hmap

splitHndsQSemR :: Handle → [(HndP, Splitter ρ_QSem)] → [(HndP, Splitter ρ_QSem)]
splitHndQSemR (QSemRdHnd h) hmap = ovwLM h (Split R) hmap
splitHndQSemR _             hmap = case (lkpLM h hmap) of
  Just x   -> hmap
  Nothing  -> ovwLM h (Split R) hmap
```

Appendix B

Additional proofs and machine-verification

B.1 Additional maximal lattice results

Define two unary operators $\triangle$ and $\triangledown$ as follows:

$$\begin{aligned} \triangledown c &\triangleq \neg\neg{\sim}c \\ \triangle c &\triangleq \neg{\sim}\neg c \end{aligned}$$

The operators $\triangle$ and $\triangledown$ have been shown to obey the following properties. As well as being proved, the vast majority of these have also been experimentally verified to hold for over 10,000 test I/O models using a small Haskell program.

$$\begin{array}{lrcl} \text{(i)} & \triangledown\neg c^+ & = & \neg\triangle c^+ \\ \text{(ii)} & \triangledown(c_1^+ \sqcap c_2^+) & = & \triangledown c_1^+ \sqcup \triangledown c_2^+ \\ \text{(iii)} & \triangle(c_1^+ \sqcup c_2^+) & = & \triangle c_1^+ \sqcap \triangle c_2^+ \\ \text{(iv)} & c^+ \sqcup \triangledown c^+ & = & \mathbb{1} \\ \text{(v)} & c^+ \sqcap \triangle c^+ & = & \mathbb{0} \\ \text{(vi)} & \triangle c^+ & \subseteq & \triangledown c^+ \\ \text{(vii)} & \triangle\triangle\triangle c^+ & = & \triangle c^+ \\ \text{(viii)} & \triangledown\triangledown\triangledown c^+ & = & \triangledown c^+ \end{array}$$

$$\begin{array}{lrcccl} \text{(ix)} & \triangledown\triangledown c^+ & \subseteq & c^+ & \subseteq & \triangle\triangle c^+ \\ \text{(x)} & \triangle\mathbb{1} & = & \mathbb{0} & = & \triangledown\mathbb{1} \\ \text{(xi)} & \triangle\mathbb{0} & = & \mathbb{1} & = & \triangledown\mathbb{0} \end{array}$$

Most of these results require separate lemmas, and these are proved over the following pages.

- (i): A direct result of the definition. $\triangledown\neg c^+ = \neg\neg{\sim}\neg c^+ = \neg\triangle c^+$

- (ii): Use the de Morgan rules for $\neg$ and $\sim$. $\nabla(c_1^+ \sqcap c_2^+) = \neg\neg\sim(c_1^+ \cap c_2^+) = \neg\neg(\sim c_1^+ \cup c_2^+) = \neg\neg\neg(\neg\sim c_1^+ \cap \neg\sim c_2^+) = \neg\neg(\neg\neg\sim c_1^+ \cup \neg\neg\sim c_2^+) = \nabla c_1^+ \sqcup \nabla c_2^+$
- (iii): Similar to the proof of (ii).
- (iv): Use Lemma B.1.5. This states that $\neg c \cup \neg\neg\sim\neg c = A$, which equals $c^+ \cup \nabla c^+ = \mathbb{1}$ which implies $\neg\neg c^+ \cup \nabla c^+ = \neg\neg\mathbb{1}$. Therefore $c^+ \sqcup \nabla c^+ = \mathbb{1}$.
- (v): Once again use Lemma B.1.5. Since $\neg c \cup \neg\neg\sim\neg c = A$, substituting $\neg c$ for c and negating the entire equation gives $\neg(\neg\neg c \cup \neg\neg\sim\neg\neg c) = \neg A$ which rewrites (using de Morgan laws and removing excess $\neg$s), to $\neg c \cap \neg\sim\neg\neg c = \neg A$. This is the same as $c^+ \sqcap \triangle c^+ = \mathbb{0}$.
- (vi): Lemma B.1.1.
- (vii): Lemma B.1.7.
- (viii): From Lemma B.1.7 (vii) it is true that $\triangle\triangle\triangle c = \triangle c$. Therefore, replacing c with $\neg c^+$ and negating both sides, this gives $\neg\triangle\triangle\triangle\neg c^+ = \neg\triangle\neg c^+$. Rewriting (i) multiple times, this gives $\nabla\nabla\nabla\neg\neg c^+ = \nabla\neg\neg c^+$, which is equivalent to $\nabla\nabla\nabla c^+ = \nabla c^+$.
- (ix): Lemma B.1.6 states that $c^+ \subseteq \triangle\triangle c^+$. The result for $\nabla\nabla c^+$ follows directly in the same style as the proof for (v).
- (x): From Lemma B.1.3 $\neg\sim\neg A = \neg A$. Therefore $\triangle\mathbb{1} = \mathbb{0}$. To prove $\nabla\mathbb{1} = \mathbb{0}$ just expand the definition: $\nabla\mathbb{1} = \neg\neg\sim\mathbb{1} = \neg\neg\emptyset = \mathbb{0}$.
- (xi): Similar to proof of (x).

Lemma B.1.1. $\triangle c \subseteq \nabla c$

Proof. First, expand the definitions of $\triangle c$ and ∇c.

$$\begin{aligned}
\triangle c &= \neg\sim\neg c \\
&= \{a \in A \mid \forall_{a_1 \in \sim\neg c}.a \odot a_1\} \\
&= \{a \in A \mid \forall_{a_1 \in A}.(a_1 \notin \neg c) \Longrightarrow a \odot a_1\} \\
\nabla c &= \neg\neg\sim c \\
&= \{a \in A \mid \forall_{a_1 \in \neg\sim c}.a \odot a_1\} \\
&= \{a \in A \mid \forall_{a_1 \in A}.(\forall_{a_2 \in \sim c}.a_2 \odot a_1) \Longrightarrow a \odot a_1\} \\
&= \{a \in A \mid \forall_{a_1 \in A}.(\forall_{a_2 \in A}.a_2 \notin c \Longrightarrow a_2 \odot a_1) \Longrightarrow a \odot a_1\}
\end{aligned}$$

Quantifying over all $a \in A$ at the outermost level this gives the following proof goal

$$(\forall_{a_1 \in A}.(a_1 \notin \neg c) \Longrightarrow a \odot a_1) \Longrightarrow (\forall_{a_1 \in A}.(\forall_{a_2 \in A}.a_2 \notin c \Longrightarrow a_2 \odot a_1) \Longrightarrow a \odot a_1)$$

Moving the quantification of the right-hand $a_1 \in A$ to the outermost level, and changing some bound names, this becomes

$$(\forall_{a_3 \in A}.(a_3 \notin \neg c) \Longrightarrow a \odot a_3) \Longrightarrow (\forall_{a_2 \in A}.a_2 \notin c \Longrightarrow a_2 \odot a_1) \Longrightarrow a \odot a_1$$

It is straightforward that the consequent $a \odot a_1$ can be proved if either $a_1 \notin \neg c$ (instantiating a_3 with a) or $a \notin c$ (instantiating a_2 with a_1). Now, suppose that this is not the case, namely that both $a_1 \in \neg c$ and $a \in c$. Then $a \odot a_1$ still holds, since $a_1 \in \neg c$ means, from the definition of $\neg c$, that $(\forall_{a_4 \in c}.a_4 \odot a_1)$, and a_4 can be instantiated with a because $a \in c$. □

Lemma B.1.2. *If $c_1 \subseteq c_2$ then $\triangle c_2 \subseteq \triangle c_1$.*

Proof. The definition of $\triangle c$ is

$$\begin{aligned} \triangle c &= \neg\sim\neg c \\ &= \{a \in A \mid \forall_{a_1 \in \sim\neg c}.a \odot a_1\} \\ &= \{a \in A \mid \forall_{a_1 \in A}.\neg(\forall_{a_2 \in c}.a_2 \odot a_1) \Longrightarrow a \odot a_1\} \\ &= \{a \in A \mid \forall_{a_1 \in A}.\forall_{a_2 \in c}.a_2 \not\odot a_1 \Longrightarrow a \odot a_1\} \end{aligned}$$

Therefore,

$$\triangle c_2 \subseteq \triangle c_1 \quad = \quad \forall_{a \in A}.(\forall_{a_1 \in A}.\forall_{a_2 \in c_2}.a_2 \not\odot a_1 \Longrightarrow a \odot a_1) \Longrightarrow (\forall_{a_3 \in A}.\forall_{a_4 \in c_1}.a_4 \not\odot a_3 \Longrightarrow a \odot a_3)$$

Moving quantifiers, this is equivalent to stating that for all $a, a_3 \in A$ and for all $a_4 \in c_1$

$$(\forall_{a_1 \in A}.\forall_{a_2 \in c_2}.a_2 \not\odot a_1 \Longrightarrow a \odot a_1) \Longrightarrow a_4 \not\odot a_3 \Longrightarrow a \odot a_3$$

Since it as assumed that $c_1 \subseteq c_2$, we know that $a_4 \in c_2$. So instantiating a_2 with a_4 and a_1 with a_3 this gives

$$(a_4 \not\odot a_3 \Longrightarrow a \odot a_3) \Longrightarrow a_4 \not\odot a_3 \Longrightarrow a \odot a_3$$

which is obviously true.

□

Lemma B.1.3. $\neg\sim\neg A = \neg A$

Proof. Prove $\subseteq$ and $\supseteq$ separately.

- $\neg\sim\neg A \subseteq \neg A$: from Lemma B.1.1 it is true that $\neg\sim\neg c \subseteq \neg\neg\sim c$, so we need only prove $\neg\neg\sim A \subseteq \neg A$, which holds since $\neg\neg\sim A = \neg\neg\emptyset = \neg A$.
- $\neg A \subseteq \neg\sim\neg A$: because for all c, $c \subseteq A$, it is true that $\sim\neg A \subseteq A$. Apply Proposition 8.1.4 to prove that $\neg A \subseteq \neg\sim\neg A$.

□

Lemma B.1.4. *If $a_1 \in c$ and $a_2 \in \neg\sim\neg c$ then for all actions $a \in A$, either $a \odot a_1$ or $a \odot a_2$.*

Proof. Action a is either an element of $\neg c$ or it is not.

- $a \in \neg c$: this means a commutes with all actions in c, so since $a_1 \in c$, $a \odot a_1$ holds.
- $a \notin \neg c$: this is the same as saying $a \in \sim\neg c$, and the definition of $a_2 \in \neg\sim\neg c$ is that a_2 commutes with all actions permitted by $\sim\neg c$. Therefore, $a \odot a_2$.

□

Lemma B.1.5. $\neg c \cup \neg\neg\sim\neg c = A$

Proof. This is a direct consequence of Lemma B.1.4 and we begin from that result.

$$\forall_{a\in A}.\forall_{a_1\in c}.\forall_{a_2\in\neg\sim\neg c}.a \odot a_1 \ \vee \ a \odot a_2$$

By moving universal quantifiers, this gives

$$\forall_{a\in A}.(\forall_{a_1\in c}.a \odot a_1) \ \vee \ (\forall_{a_2\in\neg\sim\neg c}.a \odot a_2)$$

which is equivalent, from the definition of $\neg$, to $\forall_{a\in A}.a \in \neg c \vee a \in \neg\neg\sim\neg c$. This means that every a is either an element of $\neg c$ or an element of $\neg\neg\sim\neg c$. Therefore, $\neg c \cup \neg\neg\sim\neg c = A$, the set of all actions. □

Lemma B.1.6. $c \subseteq \triangle\triangle c$

Proof. Expand the definition of $\triangle\triangle c$:

$$\begin{aligned}
\triangle\triangle c &= \neg\sim\neg\neg\sim\neg c \\
&= \{a \in A \mid \forall_{a_1\in\sim\neg\neg\sim\neg c}.a_1 \odot a\} \\
&= \{a \in A \mid \forall_{a_1\in A}.\neg(\forall_{a_2\in\neg\sim\neg c}.a_1 \odot a_2) \Longrightarrow a_1 \odot a\} \\
&= \{a \in A \mid \forall_{a_1\in A}.\forall_{a_2\in\neg\sim\neg c}.a_1 \not\odot a_2 \Longrightarrow a_1 \odot a\} \\
&= \{a \in A \mid \forall_{a_1\in A}.\forall_{a_2\in A}.(\forall_{a_3\in\sim\neg c}.a_3 \odot a_2) \ \wedge \ a_1 \not\odot a_2 \Longrightarrow a_1 \odot a\}
\end{aligned}$$

Therefore

$$\begin{aligned}
c \subseteq \triangle\triangle c &= c \subseteq \{a \in A \mid \forall_{a_1\in A}.\forall_{a_2\in A}.(\forall_{a_3\in\sim\neg c}.a_3 \odot a_2) \ \wedge \ a_1 \not\odot a_2 \Longrightarrow a_1 \odot a\} \\
&= \forall_{a\in A}.a \in c \Longrightarrow (\forall_{a_1\in A}.\forall_{a_2\in A}.(\forall_{a_3\in\sim\neg c}.a_3 \odot a_2) \ \wedge \ a_1 \not\odot a_2 \Longrightarrow a_1 \odot a) \\
&= \forall_{a\in c}.\forall_{a_1\in A}.\forall_{a_2\in A}.(\forall_{a_3\in\sim\neg c}.a_3 \odot a_2) \ \wedge \ a_1 \not\odot a_2 \Longrightarrow a_1 \odot a
\end{aligned}$$

The proof of the lemma becomes a proof that for all $a_1, a_2 \in A$ and for all $a \in c$, if $\forall_{a_3\in\sim\neg c}.a_3 \odot a_2$ and $a_1 \not\odot a_2$ then $a_1 \odot a$. This can be shown by contradiction. If $a_1 \not\odot a$, then since $a \in c$, $a_1 \notin \neg c$ (a_1 is not an action which commutes with all actions in c) which

is equivalent to $a \in \sim\neg c$. Instantiating a_3 with a_1, this lets us prove $a_1 \odot a_2$, but since it is already known that $a_1 \not\odot a_2$, this is contradictory. □

Lemma B.1.7. $\triangle\triangle\triangle c = \triangle c$

Proof. Prove two separate inequalities:

- $\triangle c \subseteq \triangle\triangle\triangle c$: a direct consequence of Lemma B.1.6, which shows that $(\triangle c) \subseteq \triangle\triangle(\triangle c)$.
- $\triangle\triangle\triangle c \subseteq \triangle c$: from Lemma B.1.6 we know that $c \subseteq \triangle\triangle c$, so apply Lemma B.1.2 to prove that $\triangle(\triangle\triangle c) \subseteq \triangle(c)$.

□

B.2 Full Abstraction and Admissibility

B.2.1 Full abstraction

The question of how the full abstraction problem for PCF affects formal reasoning is a valid one. Plotkin's original paper [98] showed how there exists a monotonic "parallel-or" function `por` in the domain $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$ which is defined as follows:

`por`	True	False	$\bot$
True	True	True	True
False	True	False	$\bot$
$\bot$	True	$\bot$	$\bot$

This function cannot be defined in PCF (or Core-Clean) since one must force the evaluation of each argument in some fixed sequential order. This restriction is immediately present in the proof assistant. Due to the fact that the denotational semantics does not quite fit the operational semantics, in Sparkle the following operationally true theorem cannot be proved (or disproved):

$$\neg\exists_{\texttt{por}\in\mathsf{Bool}\to\mathsf{Bool}\to\mathsf{Bool}}\cdot$$
$$\texttt{por}\ \textsc{True}\ \bot = \textsc{True} \ \wedge\ \texttt{por}\ \textsc{False}\ \textsc{False} = \textsc{False} \ \wedge\ \texttt{por}\ \bot\ \textsc{True} = \textsc{True}$$

Can the full-abstraction problem lead to inconsistent proofs in our proof assistant? Not as far as we know. When we prove a property $\forall_{x\in\tau}.P(x)$ we quantify over all elements of domain τ, possibly including those which have no operational equivalent, such as `por`. But when one proves a property $\exists_{x\in\tau}.P(x)$, one is not able to witness these non-existent elements such as `por` – one must specify an actual Core-Clean term. This does mean that Sparkle's logic is a little unusual. In any logic of this form universal quantification will not be the dual of existential quantification [2]. But despite the lack of formal documentation

on the proof-assistant, the author has used it extensively for a period of over two years and has never, to his knowledge, encountered a logical bug.

B.2.2 Admissibility

Admissibility, described briefly in [86] and in detail in [56], is a constraint on a proposition concerning a lazy structure which must hold before induction on that structure is sound. There are some simple checks which provide a basic but incomplete test. These are:

- $t_1 = t_2$ is admissible.
- $\forall_{x \in \alpha}.P_1$ is admissible if P_1 is admissible.
- $\forall_{x \in \alpha}.P_1$ is admissible if α is a flat domain.
- $P_1 \wedge P_2$ is admissible if P_1 and P_2 are both admissible.
- $P_1 \vee P_2$ is admissible if P_1 and P_2 are both admissible.
- $\neg P_1 \Longrightarrow P_2$ is admissible if P_1 and P_2 are both admissible.
- $t_1 \neq \bot$ is admissible.

The first two in effect state that all standard equality proofs *à la* Bird [12] are admissible. Existential predicates and implication cause problems. Consider the programs `ones`, an infinite list of `1`s, and `finite`, which returns `True` if its argument list is finite, otherwise failing.

```
ones :: [Int]
ones = (1:ones)
```

```
finite :: [a] -> Bool
finite []     = True
finite (x:xs) = finite xs
```

Without admissibility checks the following two false theorems would be easily provable by inducting over list xs.

- $\forall_{xs \in [\alpha]}.\exists_{i \in \mathbb{N}}.xs$ `!!` $i = \bot$. This is false because for all non-negative i, `ones !!` $i = 1$.
- $\forall_{xs \in [\alpha]}.xs =$ `ones` $\Longrightarrow$ `finite` $xs =$ TRUE. This is false because `finite ones` $= \bot$.

Both of the above propositions are only true for all finite, partial lists. Admissibility in effect states that a result which holds for all finite, partial structures will also hold when those structures can be infinite.

B.3 Sparkle proof sections

The machine-readable form of the proofs may be obtained from the following URL:

`http://www.cs.tcd.ie/research_groups/fmg/archive/Dowse_PhD/`

It isn't always easy to simply list the Sparkle theorem which corresponds to each individual lemma. Almost all proofs which required induction over program structure (i.e. those in Chapter 4) needed to be proved in such a way that Sparkle could show it was admissible. These results contain the guts of the proof, and usually a theorem was then proved which expressed the result in a more natural style. Also, many definedness results were omitted entirely in this document. These are uninteresting but still affect what was actually proved and our ability to form a direct link between presented lemmas and the Sparkle theorems.

The implementation presented in this document is slightly different to that which can be downloaded. Many changes are very simple, such as those relating to strictness annotation or changes in function names made to aid presentation. Others are are a little more complex. The implementations presented here are idealised ones – if we could start from scratch again we would choose these. The actual implementations can be messier, but changing them at a later stage would have been impossible without redoing all the Sparkle proofs.

A total of 497 machine-verified results were performed by the author. Not all results which could have been machine-checked were. In particular, due to a lack of time, the confluence proofs for the I/O models defined in Chapter 6 were just performed "by hand". theorems in that section. Figure B.1 contains a list of some of the more important results in this document for which there are equivalent Sparkle theorems.

Result	Sparkle Section	Theorem Name
Lemma 2.2.1	curio_examples	PRE_Buffer
Lemma 2.2.2	curio_examples	PRE_Lock
Lemma 2.2.3	curio_examples	PRE_IVar
Lemma 2.2.4	curio_examples	PRE_IStr
Figure 3.1	curio_single_step_rules	*whole section*
Lemma 3.3.1	dountil_SList	dountil_main_theorem
R.E. Proof	curio_rdce	rdce_run
Lemma 4.2.1	curio_prelude	next_separated
Lemma 4.2.8	curio_prelude	nextRedex_Par_False
Lemma 4.2.9	curio_prelude	nextRedex_Par_True
Lemma 4.3.1	curio_reduction	nextRedex_stalled_any_guess
Lemma 4.3.3	curio_reduction	nextRedex_wa
Lemma 4.3.5	curio_reduction	nextRedex_ap
Lemma 4.3.7	curio_reduction	nextRedex_interfere_Silent
Lemma 4.3.9	curio_reduction	nextRedex_interfere_Action
Theorem 4.3.1	curio_reduction	next_disjoint_with_nextRedex
Lemma 4.4.1 (i)	curio_failure_internals	length_preorder_shuffle
Lemma 4.4.1 (v)	curio_failure_internals	shuffle_mapTree
Lemma 4.4.1 (vi)	curio_failure_internals	preorder_mapTree
Lemma 4.4.1 (viii)	curio_failure_internals	firstRedex_++
Lemma 4.4.8	curio_failure	nextRedex_separated
Theorem 4.5.1	curio_failure	next_failure_divergence
Lemma 4.5.2	curio_failure	next_badlyformed_failure
Lemma 4.5.3	curio_failure	next_badlyformed_divergence
Lemma 4.6.1	curio_confluence	next_final_step
Lemma 4.6.2	curio_confluence	tidy_diamond
Theorem 4.6.1	curio_confluence	rdce_CONFLUENCE
Theorem 5.1.1	curio_combinators	IOModel_product
Theorem 5.1.2	curio_combinators	IOModel_StrMap
Theorem 5.2.1	curio_combinators	IOModel_DynMap
Theorem 5.3.1	curio_IOLocModel	PRE_MtoLM
Theorem 5.3.2	curio_IOLocModel	PRE_LMtoM
Theorem 5.3.3	curio_Loc_combinators	PRE_IOLocTimes
Theorem 5.4.1	curio_Loc_combinators	PRE_IOLocDynM
Lemma 5.3.2	curio_Loc_combinators	IOSMapN_IOLocMapN
Lemma 7.1.1	curio_equivalence	rdce_Par_left
Lemma 7.1.2	curio_equivalence	rdce_Par_left_backwards
Lemma 7.1.3	curio_equivalence	rdce_Par_right
Lemma 7.1.4	curio_equivalence	rdce_Par_right_backwards
Lemma 7.1.5	curio_equivalence	rdce_Bind_split
Lemma 7.1.6	curio_equivalence	rdce_Bind_build_left
Lemma 7.1.8	curio_equivalence	rdce_Bind_build_right
Lemma 7.1.9	curio_equivalence	Bind_wmcequiv
Lemma 7.1.11	curio_equivalence	Par_left_wmcequiv
Lemma 7.1.10	curio_equivalence_conf	Par_right_wmcequiv
Figures 7.3, 7.4	curio_big_step_rules	*whole section*

Figure B.1: Sparkle theorem names

Bibliography

[1] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[2] Samson Abramsky. private communication, May 2005.

[3] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[4] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994.

[5] Peter Achten and Rinus Plasmeijer. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, January 1995.

[6] Peter Achten and Rinus Plasmeijer. Interactive functional objects in Clean. In *IFL '97: Selected Papers from the 9th International Workshop on Implementation of Functional Languages*, pages 304–321, London, UK, 1998. Springer-Verlag.

[7] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin C. Rinard. Verifying a file system implementation. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2004.

[8] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Language Systems*, 11(4):598–632, 1989.

[9] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, Revised Edition, 1984.

[10] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[11] A. Białynicki-Birula and Helena Rasiowa. On the representation of quasi-boolean algebras. *Bulletin de L'académie Polonaise des Sciences*, 5:259–261, 1957.

[12] Richard J. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, 2nd edition, 1998.

[13] V. Boicescu, A. Filipoiu, G. Georgescu, and S. Rudeano. *Łukasiewicz-Moisil algebras*. North Holland, Amsterdam, 1991.

[14] Andrew Butterfield and Glenn Strong. Proving correctness of programs with I/O – a paradigm comparison. In Thomas Arts and Markus Mohnen, editors, *Proceedings of the 13th International Workshop, IFL2001*, volume LNCS2312, pages 72–87, 2001.

[15] Magnus Carlsson and Thomas Hallgren. FUDGETS – A graphical user interface in a lazy functional language. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 321–330, 1993.

[16] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, Department of Computing Science, March 1998.

[17] David Carter. Deterministic concurrency. Master's thesis, Department of Computer Science, University of Bristol, September 1994.

[18] Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[19] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[20] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

[21] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[22] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275, New York, NY, USA, 1999. ACM Press.

[23] Roy Crole and Andrew Gordon. A sound metalogical semantics for input/output effects. In *CSL '94: Selected Papers from the 8th International Workshop on Computer Science Logic*, pages 339–353, London, UK, 1995. Springer-Verlag.

[24] John Cupitt. A Brief Walk Through KAOS. Technical Report 58, Computing Laboratory, University of Kent, Canterbury, UK, 1989.

[25] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[26] Brian Davey and Hilary Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[27] Maarten de Mol. ?? PhD thesis. To appear., 2005.

[28] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers. In Thomas Arts and Markus Mohnen, editors, *Proceedings of the 13th International Workshop, IFL2001*, number 2312 in LNCS, pages 55–71. Springer-Verlag, 2001.

[29] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, New York, NY, USA, 2001. ACM Press.

[30] Malcolm Dowse, Andrew Butterfield, and Marko van Eekelen. Reasoning about deterministic concurrent functional I/O. In Clemens Grelck and Frank Huch, editors, *Proceedings of IFL 2004*, volume LNCS3474, pages 177–194. Springer-Verlag, 2005.

[31] Malcolm Dowse, Andrew Butterfield, Marko van Eekelen, Maarten de Mol, and Rinus Plasmeijer. Towards machine-verified proofs for I/O. Technical Report NIII-R0415, University of Nijmegen, April 2004.

[32] Malcolm Dowse, Glenn Strong, and Andrew Butterfield. Proving make correct – I/O proofs in Haskell and Clean. In Ricardo Peña and Thomas Arts, editors, *Proceedings of the IFL 2002*, volume LNCS2670, pages 68–83, 2002.

[33] Karl-Filip Faxén. A static semantics for haskell. *Journal of Functional Programming*, 12(5):295–357, 2002.

[34] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[35] Jeremy Gibbons. Calculating functional programs. In Roland Carl Backhouse, Roy L. Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 149–202. Springer, 2000.

[36] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 28–38, New York, NY, USA, 1986. ACM Press.

[37] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[38] Andrew Gordon. An operational semantics for I/O in a lazy functional language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 136–145, New York, NY, USA, June 1993. ACM Press.

[39] Andrew Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[40] Andrew Gordon. Bisimilarity as a theory of functional programming: mini-course. Notes Series BRICS-NS-95-3, BRICS, Department of Computer Science, University of Aarhus, July 1995.

[41] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, New York, 1979.

[42] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

[43] Carl Gunter. *Semantics of programming languages: structures and techniques*. MIT Press, Cambridge, MA, USA, 1992.

[44] Cordelia Hall and Kevin Hammond. A dynamic semantics for Haskell (draft), May 20 1993.

[45] Thomas Hallgren, Mark Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 116–128, New York, NY, USA, 2005. ACM Press.

[46] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[47] William Harrison, Tim Sheard, and James Hook. Fine control of demand in Haskell. In *MPC '02: Proceedings of the 6th International Conference on Mathematics of Program Construction*, pages 68–93, London, UK, 2002. Springer-Verlag.

[48] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[49] Sören Holström. PFL: A functional language for parallel programming. Technical Report Report 7, Chalmers University, Department of Computer Science, University of Aarhus, September 1983.

[50] Ian Holyer and Eleni Spiliopoulou. Concurrent monadic interfacing. In *IFL '98, 10th International Workshop, Selected Papers, London, UK, September 1998*, pages 73–89. Lecture Notes in Computer Science, Volume 1595, Springer Verlag, June 1999.

[51] Douglas Howe. Equality in lazy computation systems. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 198–203, Piscataway, NJ, USA, 1989. IEEE Press.

[52] Paul Hudak and Raman Sundaresh. On the expressiveness of purely functional I/O systems. Technical report, Yale University, 1989.

[53] Gérard Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994. Preliminary version available as INRIA Technical Report 2009, August 1993.

[54] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[55] Hugs 98. `http://www.haskell.org/hugs/`.

[56] Shigeru Igarashi. Admissibility of fixed-point induction in first-order logic of typed theories. In *Proceedings of the International Symposium on Theoretical Programming*, pages 344–383, London, UK, 1974. Springer-Verlag.

[57] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[58] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computings Survey*, 36(1):1–34, 2004.

[59] Mark Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, December 1993.

[60] Mark Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, August 1993.

[61] Simon B. Jones. A range of operating systems written in a purely functional style. Technical Monograph PRG-42, Oxford University Computing Laboratory, Oxford,

UK, 1984. Also Technical Report 16, Department of Computer Science, University of Stirling.

[62] Kent Karlsson. Nebula - A functional operating system. Technical Report LPM11, Laboratory for Programming Methodology, Chalmers University of Technology, Göteborg, SE, 1981.

[63] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[64] Peter Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[65] John Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, New York, NY, USA, 1993. ACM Press.

[66] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, Florida, June 20–24, 1994.

[67] John Lucassen and David Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM Press.

[68] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 1971.

[69] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer-Verlag, New York, 1992.

[70] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, MA, 90.1 edition, 1962.

[71] Robin Milner. Implementation and applications of Scott's logic for computable functions. *ACM SIGPLAN Notices*, 7(1):1–6, January 1972.

[72] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–22, February 1977.

[73] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[74] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, New York, NY, 1989.

[75] Robin Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England, 1999.

[76] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[77] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[78] Andrew Moran, David Sands, and Magnus Carlsson. Erratic fudgets: a semantic theory for an embedded coordination language. *Science of Computer Programming*, 46(1-2):99–135, 2003.

[79] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer-Verlag.

[80] Flemming Nielson, Hanne Riis Nielson, and Chris L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[81] Rishiyur S. Nikhil. *Id Version 88.1, Reference Manual*. MIT, Laboratory for Computer Science, Cambridge, MA, 90.1 edition, 1991.

[82] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In M. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *LNCS*, pages 733–747. Springer, 1996.

[83] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[84] C.-H. L. Ong. Correspondence between operational and denotational semantics: the full abstraction problem for PCF. *Handbook of logic in computer science (vol. 4): semantic modelling*, pages 269–356, 1995.

[85] The Programatica Project home page. `www.cse.ogi.edu/PacSoft/projects/programatica/`.

[86] Lawrence Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, New York, NY, USA, 1987.

[87] Simon Peyton Jones. Wearing the hair shirt: A retrospective on Haskell. Invited talk at POPL 2003.

[88] Simon Peyton Jones. Tackling the awkward squad – monadic input/output, concurrency, exceptions, and foreign language calls in Haskell. In CAR Hoare, M Broy, and R Stein-brueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, pages 47–96. IOS Press, 2001.

[89] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In ACM, editor, *POPL '96: Florida, 21–24 January 1996*, pages 295–308, New York, NY, USA, 1996. ACM Press.

[90] Simon Peyton Jones, Paul Hudak, Philip Wadler, and John Hughes, editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.

[91] Simon Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. *SIGPLAN Not.*, 34(5):25–36, 1999.

[92] Simon Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201, New York, NY, USA, 1989. ACM Press.

[93] Simon Peyton Jones and Philip Wadler. A static semantics for Haskell (draft), May 20 1992.

[94] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In ACM, editor, *POPL '93: Charleston, January 10–13, 1993*, pages 71–84, New York, NY, USA, 1993. ACM Press.

[95] Andrew Pitts. Lecture notes on denotational semantics, 1999. `http://www.cl.cam.ac.uk/Teaching/Lectures/dens/`.

[96] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[97] Rinus Plasmeijer and Marko van Eekelen. Concurrent Clean version 2.0 language report. `http://www.cs.kun.nl/~clean/`, December 2001.

[98] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[99] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[100] Helena Rasiowa. *An Algebraic Approach to Non-Classical Logics*, volume 78 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 1974.

[101] Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, 1998.

[102] David Schmidt. *Denotational Semantics – A Methodology for Language Development.* Wm. C. Brown Publishers, 1986.

[103] Manfred Schmidt-Schauß. FUNDIO: A lambda-calculus with letrec, case, constructors, and an IO-interface: approaching a theory of `unsafePerformIO`. Technical Report Frank-16, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, September 2003.

[104] Clara Segura and Ricardo Peña. Correctness of non-determinism analyses in a parallel-functional language. In Philip W. Trinder, Greg Michaelson, and Ricardo Peña, editors, *IFL*, volume 3145 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2003.

[105] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[106] Peter Sewell. On implementations and semantics of a concurrent programming language. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 391–405, London, UK, 1997. Springer-Verlag.

[107] Tim Sheard. Languages of the future. *SIGPLAN Notices*, 39(12):119–132, 2004.

[108] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381, London, UK, 2000. Springer-Verlag.

[109] Eleni Spiliopoulou. *Concurrent and Distributed Functional Systems.* PhD thesis, University of Bristol, Department of Computing Science, August 1999.

[110] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[111] Colin Taylor. A theory of core fudgets. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 75–85, New York, NY, USA, 1998. ACM Press.

[112] Tachio Terauchi and Alex Aiken. Witnessing side-effects. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 105–115, New York, NY, USA, 2005. ACM Press.

[113] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.

[114] The Coq development team. The coq proof assistant reference manual v7.2. Technical Report 255, INRIA, France, March 2002.

[115] Simon Thompson. Interactive functional programs: a method and a formal semantics. Technical Report 48*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, November 1987.

[116] Jerzy Tiuryn and Mitchell Wand. Untyped lambda-calculus with input-output. In *CAAP '96: Proceedings of the 21st International Colloquium on Trees in Algebra and Programming*, pages 317–329, London, UK, 1996. Springer-Verlag.

[117] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[118] David Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999. Special issue on linear logic.

[119] Ron van Kesteren, Marko van Eekelen, and Maarten de Mol. Proof support for general type classes. In Hans Wolfgang Loidl, editor, *Selected papers from the Fifth Symposium on Trends in Functional Programming (TFP 2004)*. Intellect, 2005.

[120] Arjen van Weelden and Rinus Plasmeijer. Towards a strongly typed functional operating system. In Ricardo Peña and Thomas Arts, editors, *Proc. 14th Int. Workshop on the Implementation of Functional Languages*, volume 2670 of *LNCS*, pages 215–231. Springer-Verlag, September 2002.

[121] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM Press.

[122] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

[123] John Williams and Edward Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 169–179, New York, NY, USA, 1988. ACM Press.

[124] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[125] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.