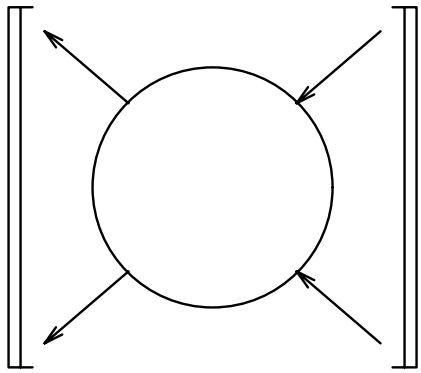
Formalising and Reasoning about Fudgets

by Colin J. Taylor

Technical Report NOTTCS-TR-98-4



Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy
December 1998

Contents

Abstract **v**

Acknowledgements **vi**

1 Introduction **1**

1.1 Functional programming 2

1.2 Reasoning about functional programs 3

1.3 Reasoning about Fudget programs 4

1.4 Results 5

1.5 Dissertation outline 6

2 Background **8**

2.1 I/O in functional programming languages 9

2.1.1 Streams 10

2.1.2 Continuation passing I/O 11

2.1.3 Monads 12

2.1.4 Uniqueness types 13

2.1.5 Semantics of functional I/O 13

2.2 Graphical development systems 14

2.2.1 Haggis 14

2.2.2 Tk-Gofer 15

2.2.3 eXene . . . 16
2.2.4 Clean . . . 17
2.2.5 Pidgets . . . 17
2.2.6 Fran . . . 18
2.3 Concurrency . . . 19

3 Fudgets **21**
3.1 Stream processors . . . 22
3.1.1 Examples . . . 23
3.1.2 Stateful stream processors . . . 24
3.1.3 Combinators . . . 25
3.1.4 Primes using stream processors . . . 26
3.2 Fudgets . . . 27
3.2.1 Layout . . . 28
3.2.2 The counter program . . . 29
3.2.3 Higher-order fudgets . . . 30
3.3 Fudget implementations . . . 31
3.3.1 Chalmers Fudgets . . . 31
3.3.2 Budgets . . . 32
3.3.3 Fudgets in Tk-Gofer . . . 33
3.3.4 Embracing Windows . . . 34
3.3.5 Fudgets in Gadgets . . . 34
3.3.6 Concurrent Haskell Fudgets . . . 35

4 Core Fudgets **36**
4.1 Types . . . 36
4.2 Terms . . . 37
4.2.1 Examples . . . 40
4.3 Encoding fudgets in Core Fudgets . . . 42

5 The π-calculus **47**

5.1 Overview 48

5.2 Syntax 49

5.3 Types 51

5.4 Semantics 54

5.4.1 Recursive process definitions 58

6 Denotational semantics of Core Fudgets **59**

6.1 Denotational semantics 60

6.2 A π-calculus semantics 60

6.3 A π-calculus semantics of literals 61

6.4 A π-calculus semantics of the λ-calculus 63

6.4.1 An Example 65

6.5 A π-calculus semantics of stream processors 66

6.5.1 Lists 67

6.5.2 Unbounded buffers 69

6.5.3 Atomic stream processors 70

6.5.4 Composite stream processors 71

7 Reasoning about Core Fudgets **76**

7.1 Bisimulation for the π-calculus 76

7.1.1 Weak bisimulations 83

7.1.2 Equivalence for Core Fudgets 84

7.2 Examples 85

7.2.1 The identity stream processor 85

7.2.2 Commutativity of parallel composition 87

7.3 Analysis of π-calculus encoding 89

7.4 Semantic issues 90

8 Operational semantics of Core Fudgets **92**

8.1 An Operational semantics 93

8.1.1 Feed 93

8.1.2 Actions 94

8.1.3 Transitions 95

8.1.4 Examples 102

8.2 A semantic theory 104

8.2.1 First-order bisimulation 106

8.2.2 Higher-order bisimulation 108

8.2.3 Examples 111

9 Axioms and applications **113**

9.1 Axioms 113

9.2 Correctness of implementations 118

10 Summary and further work **135**

10.1 Further work 136

A Proof of theorem 8.1 **138**

A.1 Overview of the proof 138

A.2 Compatible refinement 139

A.3 Compatible closure 142

A.4 Congruence of $\approx$ 144

References **163**

Abstract

The Fudgets system is a toolkit for developing graphical applications in the functional language Haskell. It models graphical objects by processes, which can be combined using a set of combinators. Although numerous programs have been developed using the Fudgets system, there has been little work upon formally reasoning about these programs. This dissertation identifies a core language underlying the Fudgets system, which we call *Core Fudgets*. Two separate semantic theories are developed for reasoning about programs written in this core langauge.

We begin by developing a denotational semantics, which considers the meaning of Core Fudget programs as π-calculus processes. We investigate the use of the theory of the π-calculus to reason indirectly about Core Fudget programs. Although this works in theory, it turns out to be awkward in practice. This can be seen as a consequence of the low-level nature of the π-calculus in comparison to the Core Fudgets language.

Using the insights gained from constructing the denotational semantics we develop an operational semantics for Core Fudgets, which considers programs as operational transitions. We develop a corresponding semantic theory based upon the concept of bisimulation, and show the associated equivalence to be a congruence. We use this theory to prove a set of equational rules useful for reasoning about Core Fudget programs. Finally, we define a notion of implementation correctness with respect to our operational semantics, and illustrate it by analysing a specific implementation.

The Core Fudgets language abstracts away from the graphical details of the Fudgets system and concentrates on the underlying communication and computation mechanisms. The theory we develop forms a foundation upon which these graphical details may be examined formally.

Acknowledgements

Firstly, my thanks go to the Languages and Programming research group in the Computer Science Department at the University of Nottingham for the facilities they have provided for my use. I would like to express my gratitude to Graham Hutton and Mark P. Jones as my supervisors for their encouragement, excellent guidance and providing me with the chance to study at Nottingham. I am also delighted to acknowledge the support of the University of Nottingham, without whose funding this research work would not have been possible.

Many thanks also to my friends in 'Lapland' for making my time studying in Nottingham a pleasurable and profitable experience. Special thanks must go to Benedict R. Gaster, Claus Reinke and Louise Dennis for the many discussions we had, and also to Anthony C. Daniels for proof-reading and for listening to my jokes.

Outside of Nottingham, my colleagues in the programming language research community have inspired and challenged me. In particular, this dissertation would not be possible without the Fudgets system developed by Thomas Hallgren and Magnus Carlsson. I thank the many people with whom I had discussions including Rob Noble, Sigbjørn Finne, Colin Runciman and Simon Peyton Jones.

Finally, I would like to thank my parents for their continued support and belief in my aims even when I was unsure of them myself.

List of Figures

3.1 Sieve of Eratosthenes 26

3.2 A Fudget 27

3.3 The counter program 29

3.4 Higher-order fudgets in action 30

3.5 A Higher-order Fudget program 31

4.1 Type rules for literals 39

4.2 Type rules for expressions 40

4.3 Type rules for stream processors 41

4.4 Type rules for constant functions 42

4.5 Serial composition for fudgets, $A >==<_F B$ 44

4.6 Encoding serial composition 44

4.7 Encoding parallel composition 45

4.8 Feedback for fudgets, **LoopF** A 45

4.9 Encoding of feedback 46

5.1 Type rules for values 52

5.2 Type rules for processes 53

5.3 An example typing derivation 54

5.4 Free and bound variables 55

5.5 Semantics for the π-calculus 56

6.1 Denotational semantics for Core Fudgets 59
6.2 An encoding of booleans 61
6.3 An encoding of natural numbers 62
6.4 An encoding of binary sums 63
6.5 An encoding of the λ-calculus 64
6.6 An encoding of type contexts 65
6.7 An encoding of lists 67
6.8 Reversing lists 68
6.9 A typing derivation for Nil 68
6.10 Buffers 69
6.11 An encoding of stream processor types 70
6.12 An encoding of atomic stream processors 71
6.13 An encoding of serial composition 71
6.14 An encoding of parallel composition 72
6.15 Splitting of streams 72
6.16 An encoding of feedback 73
6.17 An encoding of dynamic stream processors 74

7.1 An early transition semantics 79
7.2 A late transition semantics 80

8.1 The relationship between the semantics 92
8.2 The duality of **PutSP** and $\ll$ 94
8.3 Type rule for feed 94
8.4 Type rules for actions 95
8.5 An output example 102
8.6 The discard example 103
8.7 The semantics of the identity stream processor 103

8.8 A mixed input and output example 103
8.9 An example of terms not $\sim^1$ equivalent 107
8.10 Action extension rules 108
8.11 The relationship between the bisimulation equivalences 111

9.1 A representation of stream processors 120
9.2 Atomic transitions 121
9.3 The implementation of serial composition 121
9.4 Transition rules for serial composition 122
9.5 Parallel composition 123
9.6 Transition rules for parallel composition 124
9.7 The implementation of feed 125
9.8 Transition rules for feed 125
9.9 The implementation of feedback 126
9.10 Transition rules for feedback 126
9.11 The implementation of dynamic stream processors 126
9.12 Transition rules for dynamic stream processors 127
9.13 The modified implementation of serial composition 129
9.14 New serial composition transition rules 129

A.1 Compatible refinement for literals and expressions 139
A.2 Compatible refinement for stream processors 140
A.3 Compatible refinement for constant functions 141
A.4 Properties of compatible closure 142
A.5 Transitive reflexive closure 160

Chapter 1

Introduction

Numerous toolkits have been developed to construct functional programs with graphical interfaces. One particular toolkit is the Fudgets system [CH98] for the functional programming language Haskell, which models graphical objects by processes that can be combined using a simple set of combinators. Although many graphical programs have been developed using this system, there has been little work on formally reasoning about such programs. This dissertation shows that we can reason about programs written in the Fudgets system, and that this is useful for both the programmer using the system and the implementer of the system. We develop two separate theories for the Fudgets system, both of which are inspired by ideas from concurrency theory.

The first semantics we develop associates meanings to Fudget programs by a translation into the π-calculus [MPW89]. This allows us to make use of the existing theory of the π-calculus to reason about Fudget programs. Although this works in theory, it turns out to be awkward in practice. This provides the motivation for the second semantics that we develop. This second semantics associates meanings to Fudget programs by operational transitions, resulting in a more direct semantics. A corresponding theory is developed for this semantics based on the idea of bisimulation [Par81] from concurrency theory.

We examine two applications of semantic theories for the Fudgets system. Firstly, we develop some equational rules, from the theories, that can be used to reason directly about Fudget programs. Secondly, we define what it means for an implementation to be correct with respect to the theories, and we examine a specific implementation to illustrate this.

1.1 Functional programming

Functional or applicative programming languages place an emphasis on the evaluation of expressions instead of on the execution of commands. Expressions in these languages are constructed from basic values which are combined using functions.

Church's λ-calculus [Chu41] is the prototypical functional programming language. However, this was developed with the motivation of creating a calculus capturing the intuitive behaviour of functions, and was not thought of as a programming language at the time. Church's λ-notation for anonymous functions was adopted by McCarthy in the Lisp programming language [McC60] which is one of the earliest real functional languages. The next significant step in functional programming came in the ISWIM family of languages introduced by Landin [Lan66]. He introduced a number of innovations including an emphasis towards equational reasoning — reasoning about programs by replacing expressions with other equal expressions.

There are numerous modern functional languages such as Haskell [PH96], Miranda [Tur85], SML [MTH90], Lisp [McC60], and Clean [NSvEP91]. An important property of these languages is the semantics they attribute to function application. The two main ways in which function application can be interpreted are referred to as *call-by-value* semantics, and *call-by-name* semantics. In call-by-value semantics, the argument to a function is completely evaluated before the function is called. Alternatively, call-by-name semantics does not evaluate the argument before calling the function, instead when the function requires the value of the argument it is evaluated and this may occur multiple times. An *eager* functional language is one that has call-by-value semantics, whereas a *lazy* functional language has call-by-name semantics. In practice, function application in lazy languages is usually implemented with a *call-by-need* semantics which is the same as call-by-name except once an argument has been evaluated its value is stored to be recalled when the argument is next demanded. Haskell, Miranda and Clean are lazy functional languages, while Lisp and SML are eager.

In this dissertation we consider the Fudgets system which is implemented in Haskell. The concepts at the heart of this system are not restricted just to Haskell, and in our formal treatment of this system we will consider call-by-name semantics for function application. However, we believe it would be possible to modify our results for other function application semantics, such as call-by-value.

1.2 Reasoning about functional programs

An often stated advantage of functional programs is the ease with which one can reason about them. Reasoning about programs can be used to show that a program meets a specification, and also to optimise performance, space usage or other characteristics. There are many techniques available to the functional programmer to prove properties of their programs. In this section we review some of these methods including equational reasoning, induction, and coinduction.

Properties of functional programs are often proven by using *equational reasoning*, where an expression can be replaced by another that has an equal value without altering the overall meaning of the program. This technique is not just applicable to functional languages but it is usually much easier to demonstrate two expressions to have equal values for such languages. This is because most functional languages are pure, meaning that the value of a function depends only on the values of its arguments. A simple example of equational reasoning allows us to rewrite the integer doubling function as follows:

$$\begin{aligned} \mathit{double}\ x &= x + x && (\mathit{Definition\ of\ double}) \\ &= 2 * x && (\mathit{Arithmetic}) \end{aligned}$$

In this case, the resulting function might be more efficient than the original. Darlington and Burstall's classic work [BD77] on equational reasoning advocates an unfold/fold approach. A program is symbolically evaluated – unfolded – and then the resulting program is rearranged and recursion introduced – folded.

An important idea in functional programming is the recursive definition of functions. Properties can be proven of such functions by using mathematical induction on the structure of the function arguments, which in the case the arguments are of base types, such as numbers, corresponds to normal mathematical induction.

Another technique that is more suitable for proving properties of functions manipulating infinite data structures is coinduction – the dual proof principle to induction. Gordon [Gor92] advocates an operational approach to functional programming based on concurrency theory. In this approach, equivalence of terms can be defined using the concept of bisimulation [Par81] which admits coinductive proofs [MT90, Pit94]. Input and output can be integrated into this approach allowing interactive functional programs to be reasoned about.

1.3 Reasoning about Fudget programs

The Fudgets system is a toolkit for developing graphical applications in the functional language Haskell. It models graphical objects by simple processes, which can be combined using a simple set of combinators. Although many programs have been developed using the Fudgets system, there has been little work on formally reasoning about these programs. The primary goal of this dissertation is to explain the semantics of the Fudgets system, enabling properties to be proven of Fudget programs. There are two other attempts to define such a semantics that we are aware of:

A demand driven operational semantics for Fudgets: Hallgren and Carlsson [HC95] describe a simple operational semantics for the Fudgets system using rewriting rules. This semantics takes a demand driven approach, reducing the graphical objects in a Fudgets program that produce output before those that require input. Nondeterministic rewriting rules capture the concurrency of composing these graphical objects in parallel. The approach we take in this dissertation is inspired by concurrency theory and has no bias towards generating output over consuming input. The resulting semantics that we describe has more scope for concurrency and includes Hallgren and Carlsson's semantics as an instance.

A calculus for stream processors: More recently, Hallgren and Carlsson [CH98] have suggested a calculus for stream processors — the central concept in the Fudgets system. The calculus they propose is similar to the semantics we develop in this dissertation. However, their calculus is untyped and although they define an equivalence based on bisimulation they do not show it to be a congruence. They also show how the λ-calculus can be encoded in their stream processor calculus, but this requires streams to be able to carry values of different types, which contradicts one of the design principles of the Fudgets system. This is not a problem for their untyped calculus but as a result we cannot use their encoding with the semantics we consider in this dissertation.

This dissertation develops two theories of the Fudgets system. The first is based on a denotational semantics using the π-calculus, which provides a foundation for investigating some of the semantic issues in the Fudgets system. However,

this theory is complicated to use to prove properties of Fudget programs. Consequently, we develop a more practical theory based on Gordon's operational theory of functional languages, which makes reasoning about Fudget programs simpler.

We only concern ourselves with the logical behaviour of Fudget programs in this dissertation. Two Fudget programs that have the same logical behaviour but that differ in their physical appearance will be considered to be equivalent programs.

1.4 Results

The main contribution of this dissertation is to formalise the Fudgets system and to show how properties can be proven of programs written in the Fudgets system. The following list indicates specific original results:

- Identification of a core language of the Fudgets system.
- A denotational theory of the Core Fudgets language, based on a translation into the π-calculus.
- An investigation into using the denotational semantics for reasoning about Core Fudget programs.
- An operational theory of the Core Fudgets language, based on ideas from concurrency theory.
- An investigation into suitable equivalences for the operational theory, and a proof of congruence for the final equivalence we consider.
- A set of equational rules correct with respect to the operational semantics which are useful for reasoning about Core Fudget programs.
- A definition of implementation correctness with respect to the operational semantics, and an investigation into the correctness of the original Fudgets implementation.

1.5 Dissertation outline

This dissertation can naturally be broken into three parts. The first part surveys related work and introduces the Fudgets system:

Chapter 2: Background. This chapter introduces some useful background material for the rest of the dissertation. It begins by describing the problem of integrating input and output into functional programming languages, and reviews some of the solutions that have been proposed. The chapter concludes with a survey of some systems, other than the Fudgets system, for developing graphical applications in functional languages. In particular, we focus on the semantics of these systems where they exist.

Chapter 3: Fudgets. Fudgets is a system for developing graphical applications in the functional programming language Haskell. This chapter introduces the system and reviews some of the implementations of the system.

Chapter 4: Core Fudgets. We introduce the Core Fudgets language, which encapsulates the essence of the Fudgets system. This chapter defines the syntax, and type assignment relation for the Core Fudgets language, which forms the basis for theories of the Fudgets system that we consider in the later chapters.

The second part develops a formal semantics for the Core Fudgets language using the π-calculus, and uses this to reason about some Core Fudget programs:

Chapter 5: The π-calculus. The π-calculus is a process calculus designed for modelling mobile processes. This chapter defines the syntax, type assignment relation, and operational semantics of a minor variant of Milner, Parrow, and Walker's original calculus [MPW89]. This calculus forms the basis of the semantics examined in Chapter 6.

Chapter 6: Denotational semantics of Core Fudgets. We define a denotational semantics for the Core Fudgets language by translating terms into the π-calculus of Chapter 5.

Chapter 7: Reasoning about Core Fudgets. We present an overview of the theory of the π-calculus and then use this to reason about Core Fudget programs. The chapter concludes with a discussion of the relative advantages

and disadvantages of defining the semantics of Core Fudgets in terms of the π-calculus.

The third and final part develops an operational semantics for Core Fudgets and a corresponding equational theory. This theory is used to develop a set of equational rules and also to investigate the correctness of implementations of Core Fudgets:

Chapter 8: Operational semantics of Core Fudgets. In this chapter we develop an operational semantics for Core Fudgets. We investigate a number of equivalences for this semantics based upon bisimulation. The final equivalence we examine is shown to be a congruence and is used as the basis of an equational theory for Core Fudgets.

Chapter 9: Axioms and applications We explore the properties of the operational semantics developed in Chapter 8, and develop a set of equational rules for Core Fudgets. Some intuitively correct equational rules turn out to be invalid according to the equational theory of the operational semantics. We discuss some of these rules and the reasons for why they are invalid in our model. The operational semantics is also useful for checking the correctness of implementations of Core Fudgets and we conclude the chapter by describing how this can be done, illustrating it with the original Chalmers implementation of Fudgets.

Chapter 10: Summary and further work We end the dissertation by drawing conclusions and with suggestions for further work.

Chapter 2

Background

This chapter introduces some useful background material for the rest of the dissertation. First, we consider the problem of extending functional programming languages with support for input and output. Next, we review some of the systems, other than the Fudgets system, for developing graphical applications in functional programming languages. The chapter concludes with a presentation of methods for extending functional programming languages with concurrent features. This is useful as most of the systems for developing graphical applications in functional programming languages, including the Fudgets system, make use of concurrency.

Section 2.1 begins by describing the most obvious method for adding input and output to a functional programming language. Next we discuss the problems associated with this method, and the section concludes by reviewing the most widely used solutions to these problems. Section 2.2 reviews a number of systems other than the Fudgets system for developing graphical applications in functional programming languages. Specifically, we concentrate on systems using the lazy functional programming language Haskell, and discuss the formal semantics of these systems where they exist. Finally, the chapter concludes in Section 2.3, which examines methods for extending functional programming languages with concurrent constructs.

2.1 I/O in functional programming languages

Perhaps the most obvious way to add support for input and output is to use side effecting primitives similar to the functions used in imperative languages such as C [KR89]. This is by far the most widely used mechanism for I/O in functional programming languages and can be found in LISP, Scheme and Standard ML. An example of such side effecting primitives in Haskell might look like:

$$\begin{array}{lcl} \mathit{getChar} & :: & \mathit{Char} \\ \mathit{putChar} & :: & \mathit{Char} \rightarrow () \end{array}$$

The *getChar* function waits for a key to be pressed and returns the corresponding character, while $\mathit{putChar}\ c$ prints the character specified by its argument c.

Although such side effecting primitives seem to provide a direct method for adding I/O to a functional language they do not combine well with lazy functional programming languages. There are several reasons why this is the case; the first is that to use such primitives a programmer must know the details of the order of evaluation for the language. Laziness makes this more difficult to predict than for eager functional programming languages.

Normally, in a functional programming language terms either diverge or converge to some canonical value. However, in the presence of side effecting primitives this is no longer the case, because a term may perform some I/O operation and then continue as another term. This complicates reasoning about programs as there are more cases to consider when proving properties of terms. This applies for both eager and lazy functional programming languages

Finally, lazy functional programming languages with side effecting primitives have different call-by-name and call-by-need semantics. A simple example illustrates the problem:

$$(\backslash x \rightarrow x == x)\ \mathit{getChar}$$

Under call-by-name semantics two characters are read, whereas only one is read under call-by-need. This is important as call-by-name is usually implemented with a call-by-need semantics, but this is unsound if side effecting primitives are present in the language.

A number of approaches have been proposed for solving these problems of adding input and output to functional programming languages. In the following sections we review the most common of these approaches.

2.1.1 Streams

A stream is a lazy list of data objects. Miranda[1] [Tur85], Ponder [Fai85] and Hope [BMS80] use streams for I/O. In these systems, an interactive program is modelled as a function from one stream representing the input, to another representing the output. This is referred to as Landin-stream I/O, and if we consider simple teletype I/O then a program is a function mapping a stream of input characters to a stream of output characters. For example, in Haskell we might use lists to model streams, and then a program converting all its input to uppercase is written as follows:

$$\begin{array}{lcl} \mathit{uppercase} & :: & [\mathit{Char}] \rightarrow [\mathit{Char}] \\ \mathit{uppercase\ inp} & = & \mathit{map\ toUpper\ inp} \end{array}$$

The function *toUpper* is a part of the standard Haskell prelude and maps a character to its uppercase equivalent. The streams used for I/O must be lazy to allow output to be interleaved with reading input, otherwise all of the user's input would have to be read before any output could be produced.

There is another form of stream I/O, called synchronised-stream I/O, which is a generalised form of Landin-stream I/O. A synchronised-stream program is a function mapping a stream of I/O acknowledgements to a stream of I/O requests. Each request is synchronised by an acknowledgement. For example, if we define the I/O requests and acknowledgements by the datatypes:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Req} & = & \mathit{GetChar} \mid \mathit{PutChar\ Char} \\ \textbf{data}\ \mathit{Ack} & = & \mathit{ReadChar\ Char} \mid \mathit{Ok} \end{array}$$

then the example converting input to uppercase is written as the program:

$$\begin{array}{lcl} \mathit{uppercase} & :: & [\mathit{Ack}] \rightarrow [\mathit{Req}] \\ \mathit{uppercase\ acks} & = & \mathit{GetChar} : (\textbf{case}\ \mathit{acks}\ \textbf{of}\ ((\mathit{ReadChar}\ c) : \mathit{as}) \rightarrow \\ & & (\mathit{PutChar}\ (\mathit{toUpper}\ c)) : (\textbf{case}\ \mathit{as}\ \textbf{of}\ (\mathit{Ok} : \mathit{as}') \rightarrow \\ & & \mathit{uppercase\ as}')) \end{array}$$

[1] Miranda is a trademark of Research Software Ltd.

In Landin-stream I/O output is produced whenever the next value in the output stream is determined, and input occurs whenever it is demanded from the input stream. However, in synchronised I/O every I/O request is synchronised with a corresponding I/O acknowledgement. This allows any kind of I/O operation to be encoded using synchronised-stream I/O.

2.1.2 Continuation passing I/O

Continuation passing I/O was first introduced in the functional operating system Nebula [Kar81]. It uses a datatype to represent the interactive actions of a program. Considering simple teletype input and output this datatype might be:

$$\begin{array}{rcl} \textbf{data}\ \mathit{CPIO} & = & \mathit{PutChar}\ \mathit{Char}\ \mathit{CPIO} \\ & | & \mathit{GetChar}\ (\mathit{Char} \rightarrow \mathit{CPIO}) \\ & | & \mathit{Done} \end{array}$$

A value of type $\mathit{PutChar}\ c\ k$ represents the I/O action that outputs the character c and then continues as the program k. The term $\mathit{GetChar}\ f$ corresponds to an I/O action that reads an input character c and then continues as the program $f\ c$. Lastly, the term *Done* represents a program that has terminated. A program of type *CPIO* is executed by a small interpreter acting outside of the language's normal reduction mechanism. When a program is reduced to the term $\mathit{PutChar}\ c\ k$ then this interpreter actually performs the output action, and then starts evaluation of the continuation k. Similarly, a program that reduces to the term $\mathit{GetChar}\ f$ invokes the interpreter to wait for an input character. Once this input character, c, is received then evaluation restarts with the term $f\ c$. Finally, when a program reduces to the constructor *Done* then the interpreter terminates.

Continuation passing I/O works by specifying an exact order of evaluation for the I/O actions regardless of the calling convention of the underlying functional programming language [Plo75]. As a result they can be used in both eager and lazy functional programming languages as the basis of an I/O system. The name continuation passing I/O stems from the arguments k and f in the *PutChar* and *GetChar* constructs, which act as continuations in a style similar to the continuations used in denotational semantics [SW74].

2.1.3 Monads

Motivated by the work of Moggi [Mog89] and Spivey [Spi90], Wadler [Wad92, Wad90] proposed a style of functional programming based on monads that can be used to model impure 'features' such as input and output. The type system is used to distinguish between normal functional values and computations that may perform side effects. For example, a value of the monadic type $IO\ a$ is a computation that may perform some side effects returning a result of type a. All side effecting functions are embedded into this monadic type:

$$\begin{array}{lcl} \mathit{getChar} & :: & \mathit{IO}\ \mathit{Char} \\ \mathit{putChar} & :: & \mathit{Char} \rightarrow \mathit{IO}\ () \end{array}$$

Normal functional values are embedded into the monadic type using the *return* function, whilst values of the monadic type are combined using the function >>=:

$$\begin{array}{lcl} \mathit{return} & :: & a \rightarrow \mathit{IO}\ a \\ (>>=) & :: & \mathit{IO}\ a \rightarrow (a \rightarrow \mathit{IO}\ b) \rightarrow \mathit{IO}\ b \end{array}$$

A program is a value of the monadic type $IO\ ()$, and a simple example is a program that converts all input to uppercase, which is written as:

$$\begin{array}{lcl} \mathit{uppercase} & :: & \mathit{IO}\ () \\ \mathit{uppercase} & = & \mathit{getChar}\ >>= (\backslash c \rightarrow \\ & & \mathit{putChar}\ (\mathit{toUpper}\ c)\ >>= (\backslash \_ \rightarrow \\ & & \mathit{uppercase})) \end{array}$$

When a program is executed, the actions embedded in the monadic type are performed, realising the input and output. It is essential that the monadic type $IO\ a$ is an abstract datatype whose definition is not visible to the programmer. This ensures that values of the monadic type can only be constructed using the primitive functions such as *getChar* and the combinators *return* and >>=. As a result of this the monadic primitives such as *getChar* can only be used in a single threaded manner, and thus it is safe to implement them using side effecting functions. This mechanism is used in Haskell 1.3 [PH96].

2.1.4 Uniqueness types

The Clean language [NSvEP91] includes a linear type system called *Uniqueness Types [SBvEP94]* which allow programs to define functions that can perform side effects. If it can be guaranteed that the object on which a side effect is being performed is not used elsewhere then it is safe to update the object in-place. In Clean the type system is used to express when objects cannot be shared. The programmer adds a unique annotation to the types of function arguments to indicate if they must not be shared. The result type of a function can also be annotated to indicate if no sharing is introduced in the result. This mechanism enforces the single threaded use of objects. An illustrative example is the type of the function *FWriteC* that writes a character to a given file as a side-effect. Its type is:

$$:: \quad FWriteC \quad CHAR\ UNQ\ FILE \rightarrow UNQ\ FILE$$

and it takes a character and a unique file as its two arguments, returning a unique file as its result. This type specification guarantees that the file it is passed is not used elsewhere. An example of a dangerous use of the function is:

$$\begin{array}{ll} :: & Dangerous \quad UNQ\ FILE \rightarrow (UNQ\ FILE, UNQ\ FILE); \\ & Dangerous \quad file \rightarrow (FWriteC\ 'a'\ file, FWriteC\ 'b'\ file); \end{array}$$

In the body of the *Dangerous* function *FWriteC* is called twice but *file* is no longer a unique file because it is used twice, and it is therefore rejected by the typing system.

2.1.5 Semantics of functional I/O

There is a large body of work concerned with input and output in functional languages. However, very little of this work has addressed the issue of semantics for input and output. The most comprehensive work on the semantics of input and output in lazy functional languages is Gordon's dissertation [Gor92], where he develops an operational semantics and a corresponding equational theory based on Abramsky's applicative bisimulation [Abr89]. Previous to this, Hudak and Sundaresh [HS89] examined the expressiveness of purely functional I/O systems, and Thompson [Tho87] investigated streams in Miranda.

2.2 Graphical development systems

In this section we review some systems for developing graphical applications in functional languages. We describe the language features the systems make use of, such as for handling input and output, and the abstractions they support for constructing graphical applications. The review of each system concludes with an examination of any semantics that have been developed for the system.

We mainly concern ourselves with systems for the lazy functional programming language Haskell, and begin by examining the Haggis system and Tk-Gofer which are both based on Haskell. However, to provide a contrast, we next consider the eXene graphical development system written in Standard ML, and the graphical system for the Clean language. Finally, we consider Pidgets and Fran, which are more suitable for constructing reactive animations than graphical interfaces. However, it is possible to use them to build such interfaces and we examine them as representatives of a larger class of systems for reactive programming.

2.2.1 Haggis

The Haggis system [FJ95] provides support for developing graphical applications in Haskell, and is based on the Concurrent Haskell [FGJ96] extensions for concurrent processes. Instead of building a graphical program around a central event-dispatching loop, a program is modelled as a set of processes which can wait on semaphores that are set when events occur.

Input and output is handled in Haggis using the *IO* monad extended with extra operations for concurrency. Processes can be spawned, and communication between processes is supported by the use of atomically mutable state. The underlying concept in Haggis is the virtual I/O device. Interactive objects are modelled as virtual I/O devices much like the standard devices in modern operating systems. Graphical applications are built in Haggis in a compositional style.

Peyton-Jones et. al [FGJ96] describe an operational semantics for Concurrent Haskell based on ideas from concurrency theory. In their semantics, deterministic computation is separated from the nondeterministic computations that can be introduced using the concurrency extensions. This allows existing reasoning techniques such as β and η equivalence to be used unmodified for the deterministic parts of programs.

We are not aware of any formal semantics for the Haggis system in the literature, but an operational semantics is induced by the semantics for Concurrent Haskell. This would not result in a particularly abstract semantics but nevertheless could be used to reason formally about programs in the Haggis system.

2.2.2 Tk-Gofer

The Tk-Gofer GUI library [VSS96] is an extension of the functional language Gofer [Jon91] and makes use of the Tcl/Tk [Ous94] graphical user interface toolkit. Input and output is handled in Tk-Gofer using monads which provide a mechanism for communicating with Tcl/Tk.

Graphical applications are structured using objects called *widgets* which come in one of several kinds: toplevel widgets, window widgets, menu widgets, and canvas widgets. A toplevel widget acts as a container for other widgets, whilst a window widget is a graphical object such as a button. Menu widgets correspond to pull-down or pop-up menus, and canvas widgets correspond to objects such as circles or rectangles that can be placed on a canvas. Each widget has a unique identifying handle through which it can be manipulated. Tk-Gofer uses the constructor classes of Gofer to capture the common characteristics of groups of widgets in a class hierarchy. Widgets can respond to events by specifying callback functions which will be invoked when a specific event occurs. Claessen et. al[CVM97] discuss various mechanisms, including using constructor classes, to structure graphical programs in Tk-Gofer.

Although the original Tk-Gofer system did not support concurrency, version 2.0 of the system includes extensions for concurrency. These extensions are based on Concurrent Haskell and can be used to structure graphical applications as separate concurrent processes.

There is no formal semantics for Tk-Gofer, but as it uses only minimal extensions to the Gofer functional programming language then it may be possible to use standard equational reasoning techniques to prove properties of Tk-Gofer programs. This is not a very abstract method for reasoning about graphical programs, and assumes that the concurrency extensions are implemented deterministically.

2.2.3 eXene

The eXene toolkit is used for developing graphical applications in Concurrent ML [Rep91] — a higher-order concurrent language based on the Standard ML [MTH90] functional programming language. Concurrency is used for the same reason as in the Haggis system – to avoid biasing the architecture of the application towards the user interface.

Input and output is handled in eXene using side effecting functions whose order of execution is straightforward to predict because Standard ML is an eager language. Input is divided into three categories: keyboard, mouse and control. Typically, each window has a concurrent thread for each of these kinds of input, and may also have some threads for managing state. Events are distributed based on the hierarchy of windows, and so a window's thread must route events to its children. This can almost always be done using a generic router built into eXene. Graphical components are modelled using *widgets*, and a program creates a hierarchy of widgets, with the root corresponding to a top-level window in which the widgets will be realized. Widgets can be customised in three ways; the first is to parameterise them by attributes such as which font to use, or the function to be called when a specific event occurs. The next form of customisation is simple composition of widgets to form new widgets. Finally, a widget can be wrapped inside a function to alter the widget's behaviour, for example, to filter certain events such as mouse clicks and hence modify the widgets behaviour to these events.

There have been two main approaches to giving an operational semantics to Concurrent ML. The first, in the tradition of Standard ML [MTH90, MT91], is to define unlabelled transitions between programs, and Reppy [Rep92] uses this approach to give a semantics for Concurrent ML. An alternative approach taken by Ferriera et. al [FHJ95] uses ideas from concurrency theory. Labelled transitions between terms are defined which support equivalences based on bisimulation.

In a similar manner to the Haggis system, the formal semantics for Concurrent ML induces a semantics for the eXene system. However, this is not a particularly direct semantics and is most likely to be tedious for reasoning about eXene programs.

2.2.4 Clean

The Clean language includes support for constructing graphical applications using its uniqueness types. Input and output is performed on objects such as files that are offered to the programmer as abstract datatypes. Clean supports four main types of objects for input and output: the world, the file system, a file, and the event I/O system. The world acts as a container of all of the other objects. The file system is an object representing the current state of the file system, while a file is an object corresponding to an individual file. Finally, the event I/O system is an object modelling the event queue containing events such as key presses, and mouse clicks.

Clean supports four kinds of abstract devices which correspond to windows, menus, dialog boxes, and timers. A graphical program consists of a set of algebraic datatypes defining the devices for the program, along with an initial program state, and some event handling functions. The Clean system handles events in the order they occur and searches through the device definitions to find the function applicable to the event.

The semantics of the Clean language is based on term graph rewriting systems [BvEG+87], with function definitions corresponding to term graph rewriting rules. Reasoning about Clean programs thus reduces to reasoning about their corresponding graphs. As long as uniqueness types are preserved by term graph rewriting systems then this same method can be used to reason about interactive Clean programs. Barendsen et. al [BS93] prove such a result for a simple form of uniqueness typing. However, the uniqueness type system in the latest versions of Clean contain more advanced features and it is unclear as to whether this result still holds, if this is the case then interactive programs could be reasoned about by manipulating their graph forms.

2.2.5 Pidgets

Pidgets [Sch95] is a framework for programming graphical applications in the Gofer functional programming language. It attempts to unify two concepts which are normally dealt with separately: pictures and widgets. A widget is a graphical object that can react to events but has limited shapes, while a picture can be of arbitrary shape but cannot handle events.

Pidgets uses a monadic approach to deal with input and output, and introduces a monad which can be used to model objects whose values change over time. These objects can be arranged in a graph where the edges indicate functional dependencies representing constraints that must be maintained. A Pidget is itself a special case of one of these objects, where the value of the object corresponds to the picture's current appearance. The runtime system of Pidgets is written in C++ [Str97] and has itself developed into a framework for constructing graphical applications in C++ as described by Scholz et. al [SB96].

We are not aware of any work on formalising a semantics for the Pidgets system. However, like Tk-Gofer, Pidgets is implemented in Gofer and so it may be possible to make use of standard equational reasoning techniques to reason about Pidget programs. However, once again this approach is not particularly abstract and so is likely to be difficult to use in practice.

2.2.6 Fran

Fran [EH97] is a system for developing reactive animations in Haskell. The central ideas in Fran are behaviours and events. A behaviour is a value that can vary over time and may change, or react, depending on user input. An event corresponds to an arbitrarily complex condition that may depend on external events such as mouse clicks and keyboard presses, or animation parameters like the proximity of objects. Events also carry information such as the position of the mouse at the time the event occurs. An animation is described by a behaviour whose value is an image, but this can also correspond to a graphical user interface because behaviours can vary based on user input. Behaviours and events are constructed in a compositional manner. A behaviour is realized by sampling it at discrete time values, and this isolates the definition of behaviours from implementation dependent characteristics such as rendering speed.

Sage [Sag97] has experimented with general graphical user interface programming using the Fran model. He has reimplemented Fran using the temporal language underlying the Pidgets system, thus allowing Fran programs to perform arbitrary IO operations.

Elliot and Hudak [EH97] describe a formal denotational semantics for Fran. This semantics includes a proper treatment of real time that allows reasoning about events before they occur. However, this semantics is not particularly useful

for implementing Fran as it assumes that integration and detection of events can be done precisely. The semantics of recursive behaviours is defined as recursive mathematical functions which may or may not have unique solutions.

Ling [Lin97] extends this semantics to solve some of these problems, in particular, he formalises external events corresponding to user input. Also he examines the issues of recursively defined behaviours and describes a partial solution.

2.3 Concurrency

A major trend in systems for developing graphical applications using functional languages is their use of concurrency. Concurrency avoids biasing the architecture of the application towards the user interface by removing the need to structure the application around a central event-dispatching loop. In this dissertation concurrency plays a central role as it is forms one of the main principles of the Fudgets system. In this section we present an overview of work on tying concurrency and functional languages together.

The main problem in integrating concurrency and functional programming languages is in adding nondeterminism to the language while maintaining referential transparency. Burton [Bur88] attempted to solve this problem by putting the nondeterminism into data. An alternative approach taken by Milner [Mil90], was to show how functions could be encoded in the π-calculus [MPW89], thus achieving an integration of functions with concurrent processes. Meanwhile various programming languages mixing the two concepts were beginning to appear, such as CML [Rep91] and Facile [GMP89]. One of the applications for these languages is the construction of reactive and distributed systems. One of the main reasons for using these languages is that they attempt to offer the integration of the concurrency and functional paradigms in a model that allows formal reasoning about programs. Formal semantics have been developed for these languages [Rep92, FHJ95, GMP90]. More recent is the work of Boudol [Bou89] and also Jeffrey et. al [FHJ96] which introduce languages combining the λ-calculus and Milner's CCS process calculus [Mil85].

An interesting decision when integrating concurrency and functional languages is whether to make processes first-class or not. Ideally we would like processes to be first-class in just the same way that functions are first-class in the λ-calculus.

Thomsen [Tho89] investigated a higher-order form of CCS at about the same time that Milner et. al [MPW89] were developing the π-calculus which although only a first-order calculus, supports mobile processes, which can be used to encode higher-order processes. This was shown by Sangiorgi [San92] who introduced a higher-order version of the π-calculus.

Chapter 3

Fudgets

The Fudgets system [HC95, CH93] is a toolkit for building graphical applications in the lazy functional language Haskell. A program in the Fudgets system is composed of a collection of *fudgets*, which correspond closely to concurrent processes in a process network. Fudgets can communicate with one another and also with the window system, and usually correspond to graphical objects such as pushbuttons, text entry fields or windows.

This chapter introduces the Fudgets system and details some implementations of this system. Section 3.1 describes *stream processors* which form the basis for fudgets. Stream processors provide the concurrency mechanism of the Fudgets system. They correspond to concurrent processes with a single input channel and a single output channel for communication with other stream processors. Next, Section 3.2 introduces the concept of fudgets as stream processors that may also communicate with the window system, allowing them to perform graphical I/O. Throughout this chapter we describe the semantics of the Fudgets system in an informal manner, with a formal treatment being the subject of subsequent chapters. We conclude the chapter by reviewing some existing implementations of the Fudgets system including the original Chalmers implementation [CH95], Budgets [RS93], Fudgets in Tk-Gofer [CVM97], Embracing Windows [Tay96], and some concurrent implementations such as Fudgets in Gadgets [Nob95] and one based upon Concurrent Haskell [FGJ96]. These systems take different approaches to handling the concurrency in the Fudgets system, and highlight the need for a formal semantics.

3.1 Stream processors

A fudget is based upon the simpler concept of a *stream processor* [Lan65, Kah73], which unlike a fudget cannot perform graphical I/O. A stream is a list of values that can possibly go on forever, while a stream processor is a process that consumes an input stream of values to produce a corresponding output stream of values. The concept of stream processors has been widely used within computer science. For example, reactive systems, dataflow systems and specialised logic and functional programming systems are all examples of systems that have used stream processors.

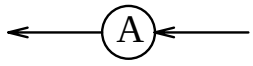
Streams were first used by Landin [Lan65] in his λ-calculus semantics of ALGOL 60 to model the values of loop variables. At about the same time Strachey [Str65] was using streams to represent input and output in his imperative GPM language. Kahn [Kah74] published an influential paper in 1974 where he modelled the semantics of process networks described by stream processors using domain theoretical techniques. He later used these ideas in joint work with MacQueen [KM77] to design a language for modelling distributed process interaction. Streams were first considered as a method for structured programming by Burge [Bur75] where he introduced a set of functional stream primitives for the purpose.

One of the first types of stream processing systems to appear in the literature are dataflow networks. Probably the most famous dataflow language is LUCID [WA85], which was introduced in 1974. This was based partly upon the POP-2 language [BCP71], which allowed a limited use of streams.

Stream processors have been used as the basis for a number of reactive systems. Specifically, the languages Signal [GGB87], LUSTRE [HCRP91] and ESTEREL [BG88], which have all been used to construct reactive systems by describing them as a set of stream processors.

Stream processors can be readily represented in functional languages using λ-abstractions to construct functions mapping input streams to output streams. Several specialised functional languages oriented towards programming with stream processors have been developed including ARCTIC [Dan84], and RUTH [Har87]. Another example of a functional approach to stream processing is the FOCUS project [BDD$^+$93] that provides a functional framework for specifying distributed systems based on stream communication.

In this dissertation we will draw stream processors as follows, with input flowing in from the right and output being produced at the left:



Streams are considered to only ever carry values of a single type, and Haskell's type system is used to ensure this invariant is always true. A stream processor of type *SP in out*, can consume values of type *in* and produce values of type *out*.

The behaviour of a stream processor is specified by the sequence in which it reads input from its input stream and produces output on its output stream. There are numerous mechanisms for accomplishing this in a functional language, such as continuation passing style I/O [Kar81], or by using monads [Wad92]. The original Fudgets system adopts a continuation passing style and defines three Haskell functions for constructing stream processors:

$$\begin{array}{lcl} putSP & :: & out \rightarrow SP\ in\ out \rightarrow SP\ in\ out \\ getSP & :: & (in \rightarrow SP\ in\ out) \rightarrow SP\ in\ out \\ nullSP & :: & SP\ in\ out. \end{array}$$

The first function produces output, the second consumes input and the third does neither. The term *putSP v k* is a stream processor that produces the value v on its output stream and then continues as the stream processor k. The stream processor *getSP f* reads a value x on its input stream and then continues as the stream processor $f\ x$. The function f is usually written as an anonymous function using the Haskell notation $\backslash x \rightarrow e$ corresponding to the lambda abstraction $\lambda x.e$. Finally, the term *nullSP* corresponds to a terminated stream processor that does not produce output or consume input.

3.1.1 Examples

A simple example of a stream processor is the identity stream processor. This reads a value on its input stream, outputs the value unchanged on its output stream, and continues as itself:

$$\begin{array}{lcl} idSP & :: & SP\ io\ io \\ idSP & = & getSP\ (\backslash x \rightarrow (putSP\ x\ idSP)). \end{array}$$

The Fudgets system provides a large library of useful stream processing functions. For example, standard list processing functions such as *map* and *filter* have corresponding stream processor versions:

$$
\begin{array}{lcl}
mapSP & :: & (in \to out) \to SP\ in\ out \\
mapSP\ f & = & getSP\ (\backslash x \to putSP\ (f\ x)\ (mapSP\ f))
\end{array}
$$

$$
\begin{array}{lcl}
filterSP & :: & (a \to Bool) \to SP\ a\ a \\
filterSP\ p & = & getSP\ (\backslash x \to \textbf{if}\ (p\ x)\ \textbf{then}\ putSP\ x\ (filterSP\ p) \\
 & & \qquad\qquad\qquad\qquad\ \textbf{else}\ filterSP\ p).
\end{array}
$$

Sometimes it is useful to be able to output a list of values from a stream processor rather than a single value, this can easily be accomplished by the *puts* function. This takes a list of values and returns a stream processor that emits these values on its output stream:

$$
\begin{array}{lcl}
puts & :: & [out] \to SP\ in\ out \to SP\ in\ out \\
puts\ [\,]\ k & = & k \\
puts\ (v : vs)\ k & = & putSP\ v\ (puts\ vs\ k).
\end{array}
$$

3.1.2 Stateful stream processors

It is sometimes useful for stream processors to have internal state. This can be accomplished by defining a stream processor recursively with an argument corresponding to the state. For example, a stream processor that stores any input it receives on its input stream in its internal state, and emits the old value of the state on its output stream can be defined as:

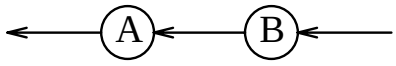
$$
\begin{array}{lcl}
state & :: & s \to SP\ s\ s \\
state\ s & = & getSP\ (\backslash x \to putSP\ s\ (state\ x)).
\end{array}
$$

In this case the stream processor behaves the same regardless of the value of the state. However, if the behaviour depends on the value of the state then instead of defining the stream processor as a large case statement then a set of stream processors could be used to model the corresponding finite state machine.

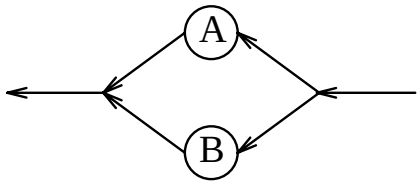
3.1.3 Combinators

Stream processors are combined together to construct a network of stream processors that operate concurrently. Three different combinators are provided by the Fudgets system for joining stream processors together:

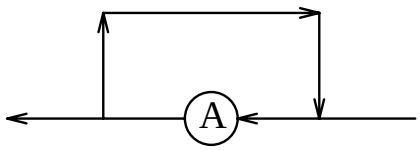
- *Serial composition*, connects two stream processors in series. In Haskell we write $A >==< B$ to denote the serial composition of the stream processors A and B. The output stream of the stream processor B is connected to the input stream of the stream processor A:



- *Parallel composition*, connects two stream processors in parallel. In Haskell we write $A >*< B$ to denote the parallel composition of the stream processors A and B. Any input to the composition is routed to both of the stream processors via their input streams, while output produced by the stream processors is merged to form a single output stream. Hallgren and Carlsson [HC95] define this merge operation to be nondeterministic, and we will interpret it in the same way. However, a practical implementation will most likely merge the outputs in chronological order. A parallel composition of stream processors is depicted as:



- *Feedback*, connects a single stream processor in a loop, and in Haskell we write *LoopSP A* to denote this. The output from the stream processor A is fed back into itself via its own input stream as well as being produced on the output stream of the overall loop. This routing of streams is depicted as:



Note that two streams are merged on the output side in parallel composition, while the feedback combinator merges two streams on the input side.

3.1.4 Primes using stream processors

As an example of a stream processing program we consider generating prime numbers using the Sieve of Eratosthenes [Mat93]. The Haskell code in Figure 3.1 implements this algorithm using stream processors. The stream processor *integers n* produces the list of integers starting from n on its output stream. We remove non-prime numbers from this list using the *filter p* stream processor, which reads integers from its input stream and only outputs the subset of these integers that cannot be factored by p. The stream processor *sift* expects to receive a prime number on its input stream which it sends to its output stream. It then continues as a stream processor that sifts out numbers that can be factored by this prime number using a serial composition of itself and *filter*. Finally, the whole prime number program, *sieve*, is constructed by using *sift* and supplying it with the entire list of integers starting at the first prime number.

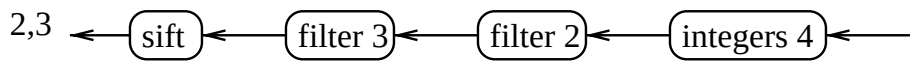
```
integers     :: SP a Int
integers n   =  putSP n (integers (n + 1))

filter       :: Int → SP Int Int
filter p     =  getSP (\n → if n `mod` p ≠ 0
                            then putSP n (filter p)
                            else filter p)
sift         :: SP Int Int
sift         =  getSP (\p → putSP p (sift >==< filter p))

sieve        :: SP a Int
sieve        =  sift >==< integers 2
```

Figure 3.1: Sieve of Eratosthenes

For each newly discovered prime number, an extra stream processor is created by *sift*. This new stream processor is an instance of *filter* whose task is to remove all multiples of the newly discovered prime number. A serial composition of these *filter* stream processors builds up forming the sieve. After producing the first two prime numbers the stream processor network looks like:



3.2 Fudgets

A fudget is a stream processor that has two extra streams for communicating with the window system. These extra streams are referred to as low-level streams, while the input and output streams inherited from stream processors are called high-level streams. The type system is used to capture the types of the high-level streams, and a fudget of type *F in out* can receive values of type *in* on its high-level input stream, and can produce values of type *out* on its high-level output stream. The types of the low level streams correspond to the types used for window requests and window responses which are fixed and therefore do not need to appear in the type for fudgets. Figure 3.2 illustrates the various streams of a fudget.

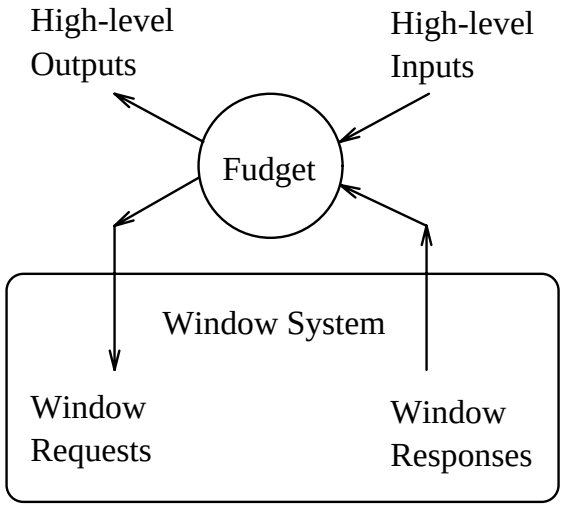


Figure 3.2: A Fudget

Fudgets can be composed in series, parallel and by using feedback similarly to stream processors using the combinators $>==<_F$, $>*<_F$ and **LoopF** . These combinators compose the high-level streams in exactly the same way as for stream processors. Instead of having many low-level streams connected to the window system, the original Fudgets system only connects the two low-level streams of the overall program to the window system. The low-level streams of the fudgets composing the overall program must be routed together to allow any fudget to send messages to the window system and also to receive responses from the window system. This routing is taken care of by the fudget versions of the combinators.

Fudgets that do not make use of their low-level streams, are called *abstract fudgets*. These are most often specified as stream processors that are converted into fudgets by the following function:

$$absF \;\; :: \;\; SP \; in \; out \rightarrow F \; in \; out.$$

The Fudgets system includes a large library of fudgets encapsulating most graphical components useful in the interfaces of programs. However, the user can extend this library either by composing existing fudgets together, or by using equivalents of the stream processor functions *getSP* and *putSP* extended to work with both the high and low level streams.

A fudget is turned into a useful Haskell program by using a top-level function, that converts a fudget into an interaction with the environment:

$$fudlogue \;\; :: \;\; F \; in \; out \rightarrow Dialogue.$$

The high-level input and output streams of the fudget supplied to this function are ignored as all I/O is achieved through the way in which the fudget interacts with the window system over its low-level streams.

3.2.1 Layout

The fudgets programs we have considered so far state nothing about where the constituent fudgets appear on screen relative to one another. The Fudgets system uses a default layout mechanism to decide where to put fudgets on the screen, and this is useful as we do not need to think about layout while developing a program. However, eventually we will want to control this layout and the Fudgets system provides three ways for doing this:

- *Placer Layout*, this method uses functions that modify the layout of a single fudget. Because fudgets are combined using the fudget combinators, this may alter the layout of many fudgets.
- *Combinator Layout*, this method extends the fudget combinators to include information about the layout of a Fudgets program. However, the flexibility in the layouts possible is constrained by the flow of data in the program because the fudget combinators control the flow of data.

- *Name Layout*, this method specifies the layout of fudgets independently from the specification of the flow of data between fudgets. Fudgets are named, and the layout specified in terms of these names, resulting in a more flexible mechanism than combinator layout.

We will not be concerned with the layout of Fudget programs in this dissertation, as our main concern is the interactive behaviour of Fudget programs. Two programs that have the same logical behaviour but differ in their physical appearance will be considered to be equivalent programs.

3.2.2 The counter program

A simple example of a complete Fudget program is the counter program. This program has a pushbutton labelled "Increment", and a text field that displays a number which is initially 0. When the user presses the pushbutton, the value displayed in the text field is incremented by one. Using the Fudgets system the Haskell code for this program is as listed in Figure 3.3.

```
main        :: Dialogue
main        =  fudlogue (shellF "Counter" counterF)

counterF    :: F Click Click
counterF    =  intDispF                          >==<F
               absF (putSP 0 (countSP 0))        >==<F
               buttonF "Increment"

countSP     :: Int → SP Click Int
countSP n   =  getSP (\c → let n' = n + 1 in putSP n' (countSP n'))
```

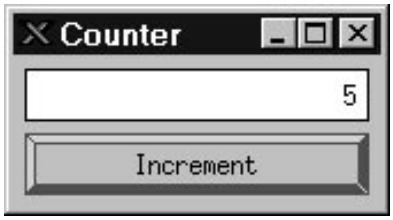


Figure 3.3: The counter program

The function *intDispF* creates an integer display fudget. Whenever an integer value is read by this fudget on its high-level input stream, it updates the integer displayed in its corresponding text field on screen to match the integer read. The state of the counter is maintained by the abstract fudget *countSP*, whose initial state is set to 0. The pushbutton is created by *buttonF*, which is a fudget that emits a *Click* value on its high-level output stream whenever the user clicks on the pushbutton. The overall interface is contained in a top-level window which is created by the *shellF* fudget, whose first argument is the title of the window and whose second argument is the fudget to be displayed inside of this window.

3.2.3 Higher-order fudgets

Fudgets are first-class objects in the Fudgets system, as they can be sent and received by fudgets themselves. This is useful as it allows interfaces which change dynamically to be constructed. A special combinator is provided for this purpose:

$$dynF \;\; :: \;\; F\ in\ out \rightarrow F\ (Either\ (F\ in\ out)\ in)\ out.$$

The first argument to this combinator defines the initial behaviour of the overall fudget. The *Either* datatype is a binary sum type and is used to classify input to the fudget as either a new fudget that should replace the current one, or as normal input values.

A simple example of the use of higher-order fudgets is shown in Figure 3.4. This program initially consists of a single button, but when this is pressed a second button is created dynamically. The source code for this program is shown in Figure 3.5.

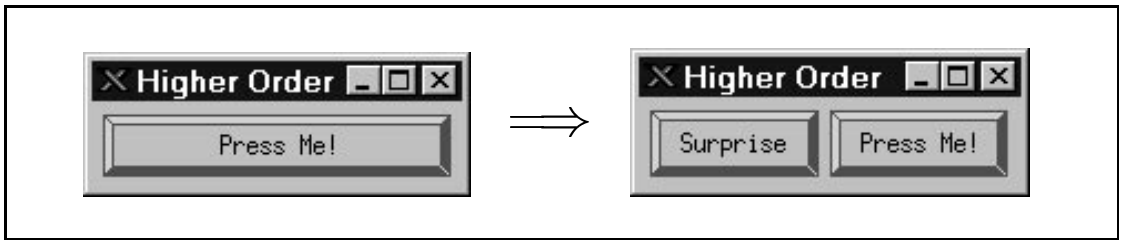


Figure 3.4: Higher-order fudgets in action

```
main           :: Dialogue
main           =  fudlogue (shellF "Higher Order" higherOrderF)

higherOrderF   :: F Click Click
higherOrderF   =  dynF (absF nullSP)    >==<F
                  absF newSP            >==<F
                  buttonF "Press Me!"

newSP  :: SP Click (Either (F Click Click) b)
newSP  =  getSP (\c -> putSP (Left (buttonF "Surprise")) nullSP)
```

Figure 3.5: A Higher-order Fudget program

3.3 Fudget implementations

In this section we review some of the existing implementations of the Fudgets system. Specifically, we concern ourselves with the variations in implementation techniques, and the level of concurrency of the systems. This is useful as it illustrates the wide variety of different implementations and techniques, and demonstrates the need for an implementation independent formal semantics.

3.3.1 Chalmers Fudgets

The original implementation of the Fudgets system, developed at Chalmers University of Technology, provides an interface to the X-Window system [SG86] from Haskell. This implementation is completely deterministic and represents stream processors by an abstract datatype, which corresponds very closely to the functions for constructing stream processors:

```
data SP inp out  =  PutSP out (SP inp out)
                 |  GetSP (inp -> SP inp out)
                 |  NullSP.
```

A stream processor that has terminated is represented by *NullSP*, while the *PutSP* constructor corresponds to a stream processor that can perform an output and then continue as some other stream processor. The *GetSP* constructor corresponds to a stream processor that can read some input.

Composing two stream processors in series, parallel and using feedback is accomplished by Haskell combinators. The design of these combinators take a productive approach to evaluation, reducing stream processors that produce output before those that require input. The inherent nondeterminism of parallel composition is eliminated by biasing towards the left branch.

Fudgets are implemented in terms of stream processors, and this encoding is described in detail in the next chapter, where we use it to describe arbitrary fudgets in terms of a core stream processing language. Originally the Chalmers implementation used synchronised streams as its I/O model, but more recently has been extended to use monadic I/O.

A large part of this implementation corresponds to a library of fudgets corresponding to common graphical components, such as pushbuttons and text entry fields. These components are all built from scratch, and make no use of the graphical components in widget sets such as Openlook [Inc90] and Motif [Ber91].

3.3.2 Budgets

The Budgets system [RS93] attempts to provide an implementation of the Fudgets system supporting the graphical components in widget sets, rather than building similar ones from scratch. The implementation is completely deterministic, and takes the same scheduling decisions as the Chalmers implementation.

In the Budget system, every fudget has an associated widget which must be created when the fudget is created. Widgets are arranged in a hierarchical fashion, and a parent widget must be specified to create one. Widgets communicate with fudgets by executing callbacks, which are blocks of code that are invoked when a particular event, such as a mouse click, occurs. The Budget system considers communication between fudgets in the same manner. On creation, a fudget is passed a special callback for communicating with other fudgets, and returns a callback that can be used to send input to the fudget. These special callbacks are called handlers, and correspond to functions taking a single argument and whose result is an I/O computations:

$$\textbf{type}\ \mathit{Handler}\ a \quad = \quad a \rightarrow \mathit{IO}\ ().$$

Monadic I/O is used to allow these handlers to perform I/O operations such as drawing in windows.

A fudget is represented as a function taking a parent widget and an output handler, and returning an I/O computation whose result is an input handler. Again, monadic I/O is used as the creation of a fudget will most likely involve some form of I/O operation, such as creating an instance of a widget.

$$\textbf{type}\ F\ in\ out\ \ =\ \ Widget \rightarrow Handler\ out \rightarrow IO\ (Handler\ in).$$

It is easy to code fudgets corresponding to the widgets of a widget set. For example, a fudget corresponding to a button widget would use a monadic primitive to create the actual button using the output handler as an argument such that it would be invoked whenever the button is pressed. The input handler that is returned could programmatically press the button by directly invoking the output handler.

Composition of fudgets involves connecting together input and output handlers appropriately. For example, the serial composition $A >==<_F B$ involves supplying the input handler returned by fudget A to fudget B as its output handler.

The Budget system does not include stream processors, instead it supports a stateful form of fudget that can be used in most places where a stream processor would be used. Stateful fudgets do not perform any graphical I/O, but make use of the mutable variable extensions to the IO monad to store the state.

3.3.3 Fudgets in Tk-Gofer

Claessen, Vullinghs, and Meijer [CVM97] describe an implementation of the Fudgets system in the Tk-Gofer system, using similar ideas to the Budgets system [RS93]. Fudgets are represented as functions using handlers, and returning monadic values. Instead of using monadic I/O to communicate directly with the window system, it provides an interface to the graphical toolkit TK [Ous94], which performs all the underlying graphical operations.

Stream processors are considered in the Fudgets in Tk-Gofer system, and are implemented purely functionally. No combinators are provided for composing stream processors, instead a coercion function is available for converting stream processors into fudgets. The fudget combinators can then be used to compose the coerced stream processors. This is not a cumbersome process because the overloading mechanism of Gofer is used to automatically insert the appropriate coercions.

This implementation is again completely deterministic and takes exactly the same scheduling decisions as the implementations in the previous two sections.

3.3.4 Embracing Windows

The Embracing Windows system [Tay96] is an extension of the Hugs [Jon95] functional programming system. Communication with the window system is supported by primitive functions embedded into the I/O monad. A framework for constructing graphical applications is developed using monadic I/O. The main objective of this system was to consider how the features of functional programming languages can be used to structure the lower-levels of a graphical framework.

A subset of the Fudgets system is built upon this framework, and uses the ideas of the Budgets system, representing fudgets by monadic functions using handlers. Stream processors are supported as a separate datatype in the style of the original Chalmers implementation. Stream processors can be combined together without being coerced into fudgets.

3.3.5 Fudgets in Gadgets

The Gadgets system [Nob95] is a concurrent extension of the functional language Gofer [Jon94]. Processes can communicate over typed channels, and can receive messages from multiple sources in an indeterminate order. Noble describes a graphical system based upon this concurrent extension, and also a *Fudget Simulator*.

A fudget is represented in this Fudget simulator as a concurrent process with two typed channels, one for input and one for output. The fudget combinators are encoded using the Gadgets parallel composition operator for processes, and by connecting processes together using channels in an appropriate fashion.

Similarly to the Fudgets in Tk-Gofer system, stream processors are supported but do not have composition combinators. Instead, they must be coerced to be fudgets and the fudget combinators used.

The scheduling of the processes corresponding to fudgets is performed by the Gadgets scheduler which is pre-emptive. As such this yields an implementation of fudgets which makes fundamental use of concurrency unlike the implementations described in the previous sections.

3.3.6 Concurrent Haskell Fudgets

Concurrent Haskell [FGJ96] is a concurrent extension of Haskell that allows it to express concurrent applications explicitly. There are two main concepts in Concurrent Haskell, processes, and atomically-mutable state. Processes are allowed to have side effects, such as altering shared state between processes. This requires that processes be treated specially, and are embedded into the IO monad to isolate the side effecting behaviour from the non-side effecting parts of the functional language. Atomically-mutable state forms the basis of all interprocess communication in Concurrent Haskell. A type is introduced to capture the type of a mutable location that can either be empty or contain a single value of a particular type. Three primitive operations can be used to manipulate such mutable locations, corresponding to creating new locations, reading the contents of locations, and writing a value into a location. A number of standard abstractions for communication between processes can be constructed using this form of atomically-mutable state. One particularly useful abstraction is an unbounded buffer called a channel.

A similar approach to the Fudget simulator described by Noble for Gadgets can be taken to develop a Fudgets system based upon Concurrent Haskell. The main idea is to represent a stream processor as a function that expects to receive two channel arguments identifying its input and output streams:

$$type\ SP\ in\ out \quad = \quad Channel\ in \rightarrow Channel\ out \rightarrow IO\ ().$$

The stream processor combinators correspond to creating new channels to connect the stream processors along with creating the stream processors as concurrent processes. For example, serial composition is encoded as the following Haskell code:

$$\begin{array}{lcl} (>==<) & :: & SP\ mid\ out \rightarrow SP\ in\ mid \rightarrow SP\ in\ out \\ (f >==< g)\ in\ out & = & \textbf{do}\ mid \leftarrow newChan \\ & & \quad fork\ (g\ in\ mid) \\ & & \quad fork\ (f\ mid\ out). \end{array}$$

This approach has been used by the author to implement the stream processing subset of the Fudgets system using Concurrent Haskell.

Chapter 4

Core Fudgets

This chapter introduces the *Core Fudgets* language, which encapsulates the essence of the Fudgets system. This language is an extension of the simply-typed λ-calculus [Chu40] and contains ground types such as booleans, and natural numbers, as well as a type for stream processors; the remaining type constructs are for functions and binary sums. The Core Fudgets language forms the basis for the theories of the Fudgets system examined in later chapters.

Section 4.1 begins by introducing the types of the language. Next, we present the terms and type assignment relation in Section 4.2. Although the Core Fudgets language is very similar to the language of the Fudgets system it does not include any concept of a fudget. However, we conclude this chapter by showing how fudgets can be encoded in terms of stream processors thus demonstrating the sufficiency of the Core Fudgets language as the basis for theories of the Fudgets system.

4.1 Types

We maintain a type system in the Core Fudgets language, primarily because the original Fudgets language is typed. The type system also ensures that only values of one type can be communicated along a particular stream. It also simplifies a formal definition of the semantics by eliminating the need to consider ill-formed terms. For example, an ill-formed term such as a stream processor expecting to receive a natural number but actually sent a character, will not be well-typed and thus is not considered by the semantics we present in Chapters 6 and 8.

We only consider a monomorphic type system — polymorphism being an orthogonal issue for our semantics — with the type expressions of our language defined by:

$$
\begin{array}{rcll}
\tau & ::= & \textbf{bool} & \textit{Boolean Type} \\
 & | & \textbf{num} & \textit{Natural Number Type} \\
 & | & \tau \to \tau' & \textit{Function Types} \\
 & | & \tau + \tau' & \textit{Disjoint Sum Types} \\
 & | & \textbf{SP}\ \tau\ \tau' & \textit{Stream Processor Types}
\end{array}
$$

We have two base types, **bool**, and **num**, which are simply examples of useful base types and more could easily be included. A disjoint sum type is included to type the combinator for dynamic stream processors, which is similar to the *dynF* combinator. We cannot encode the base types and sum types as is usually done by using functions since this would require polymorphism, and so we keep them explicit. This also makes our examples more readable. We will use $\tau, \tau', \tau'', ...$ to range over type expressions.

4.2 Terms

We assume a given countable set of term variables, $x, y, ... \in \mathit{Var}$, and define the expressions of Core Fudgets by the abstract syntax:

$$
\begin{array}{rcll}
E, F, ... \in \mathit{Expr} & ::= & S & \textit{Stream Processors} \\
 & | & C & \textit{Constant Functions} \\
 & | & L & \textit{Literals} \\
 & | & x & \textit{Variables} \\
 & | & \lambda x.E & \textit{Abstractions} \\
 & | & E\ F & \textit{Applications} \\
 & | & \textbf{Fix}\ x.E & \textit{Recursion}
\end{array}
$$

Expressions can be stream processor expressions, S, constant functions, C, or predefined values called literals, L. The usual terms of the λ-calculus are also included in this syntactic category. Recursion is supported by a fixpoint construct, **Fix** $x.E$, where x may occur free in E.

There are two categories of stream processor expressions, atomic stream processors and composite stream processors:

$$\begin{array}{lcll} S, T, ... \in SP & ::= & S_A & \textit{Atomic Stream Processors} \\ & | & S_C & \textit{Composite Stream Processors} \end{array}$$

Atomic stream processors have terms corresponding to the functions for constructing stream processors, and are defined by the abstract syntax:

$$\begin{array}{lcll} S_A, T_A, ... \in SP_A & ::= & \textbf{NullSP} & \textit{Terminated Stream Processors} \\ & | & \textbf{GetSP } E & \textit{Input Stream Processors} \\ & | & \textbf{PutSP } E\ F & \textit{Output Stream Processors} \end{array}$$

There are four forms of composite stream processors, corresponding to parallel composition, serial composition, feedback, and dynamic composition. The dynamic composition of a stream processor constructs a stream processor whose behaviour can be changed dynamically by sending it a new stream processor. This construct is essentially a stream processor version of the *dynF* combinator. Composed stream processor expressions are defined by the abstract syntax:

$$\begin{array}{lcll} S_C, T_C, ... \in SP_C & ::= & E >\!*\!< F & \textit{Parallel Composition} \\ & | & E >\!=\!=\!< F & \textit{Serial Composition} \\ & | & \textbf{LoopSP } E & \textit{Feedback} \\ & | & \textbf{DynSP } E & \textit{Dynamic Composition} \end{array}$$

There is a small collection of constant functions. These consist of a representative sample of functions for manipulating literals and could easily be extended. Also in this syntactic category are constructs for constructing and deconstructing values of sum types. The constant functions are defined by the abstract syntax:

$$\begin{array}{lcll} C, D, ... \in Const & ::= & \textbf{Left } E & \textit{Left Sum Injections} \\ & | & \textbf{Right } E & \textit{Right Sum Injections} \\ & | & \textbf{Case } E \rightarrow F, G & \textit{Sum Deconstructions} \\ & | & \textbf{if } E \textbf{ then } F \textbf{ else } G & \textit{Conditionals} \\ & | & \textbf{add } E\ F & \textit{Additions} \\ & | & \textbf{equal } E\ F & \textit{Number Equality} \end{array}$$

Finally, the syntactic category of literals is defined by the abstract syntax:

$$
\begin{array}{rcll}
L, M, ... \in \mathit{Lit} & ::= & \mathbf{true} & \textit{True literal} \\
& | & \mathbf{false} & \textit{False literal} \\
& | & \mathbf{0} & \textit{Zero Literal} \\
& | & \mathbf{1} & \textit{One Literal} \\
& | & ... &
\end{array}
$$

In the terms $\lambda x.E$, and $\mathbf{Fix}\ x.E$ the occurrence of x is a binding occurrence with the scope of x being E. An occurrence of a variable y in a term is free if it does not lie within the scope of a binding occurrence of y. The set of free variables in the term E is denoted as $FV(E)$. We define substitution in the standard manner, and use $E[F/x]$ to denote substituting F for all free occurrences of x in E, where bound variables may be renamed to avoid capture of free variables. Two terms are α-equivalent, $\equiv_\alpha$, if they only differ in the names of their bound variables.

A typing context, , , is a mapping from variables to types. A typing judgement, , $\vdash E : \tau$, asserts that term E has the type τ under the context , . If the typing context is omitted then the expression is closed. We only consider well-typed terms, defined inductively by the typing rules in Figures 4.1, 4.2, 4.3, and 4.4. The standard substitution lemma holds as follows:

Lemma 4.1 (Substitution) *If* , , $x : \tau \vdash E : \tau'$ *and* , $\vdash F : \tau$ *then it is also the case that* , $\vdash E[F/x] : \tau'$.

Proof. *A simple induction on the structure of* E. ∎

$$
\begin{array}{ll}
(\mathit{True}) & , \vdash \mathbf{true} : \mathbf{bool} \\
(\mathit{False}) & , \vdash \mathbf{false} : \mathbf{bool} \\
(\mathit{Num}) & \dfrac{\mathbf{n} \in \{\mathbf{0}, \mathbf{1}, ...\}}{, \vdash \mathbf{n} : \mathbf{num}}
\end{array}
$$

Figure 4.1: Type rules for literals

$$(Var) \qquad \frac{(x : \tau) \in ,}{, \vdash x : \tau}$$

$$(Abs) \qquad \frac{, , x : \tau \vdash E : \tau'}{, \vdash \lambda x.E : \tau \to \tau'}$$

$$(App) \qquad \frac{, \vdash E : \tau \to \tau' \quad , \vdash F : \tau}{, \vdash E\ F : \tau'}$$

$$(Fix) \qquad \frac{, , x : \tau \vdash E : \tau}{, \vdash \mathbf{Fix}\ x.E : \tau}$$

Figure 4.2: Type rules for expressions

4.2.1 Examples

In this section we illustrate the Core Fudgets language with some example programs. It is relatively straightforward to express most of the stream processors that we have already encountered as Core Fudgets programs. For example, the identity stream processor is written as:

$$\emptyset \vdash \mathbf{Fix}\ idSP.(\mathbf{GetSP}\ (\lambda x.\mathbf{PutSP}\ x\ idSP)) : \mathbf{SP\ bool\ bool}.$$

Note that we have given the type as a stream processor from booleans to booleans, although there is actually an infinite family of similar derivations with the type $\mathbf{SP}\ \tau\ \tau$, where τ is some Core Fudgets type. A polymorphic type system would allow us to infer this more general type for the term but because we only consider a monomorphic type system then we have to specify which of the infinite family of derivations we mean.

The *integers* function from Figure 3.1, which outputs increasing integers starting from a specified integer can be written in the Core Fudgets language as:

$$\emptyset \vdash \mathbf{Fix}\ integers.(\lambda n.\mathbf{PutSP}\ n\ (integers\ (\mathbf{add}\ n\ 1))) : \mathbf{num} \to \mathbf{SP\ bool\ num}.$$

The type of the input stream here is specified as carrying boolean values. However, this is only one of an infinite family of derivations which have the more general type $\mathbf{num} \to \mathbf{SP}\ \tau\ \mathbf{num}$, where τ is some type.

$$(\mathit{Null}) \qquad , \vdash \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'$$

$$(\mathit{Input}) \qquad \frac{, \vdash E : \tau \to \mathbf{SP}\ \tau\ \tau'}{, \vdash \mathbf{GetSP}\ E\ : \mathbf{SP}\ \tau\ \tau'}$$

$$(\mathit{Output}) \qquad \frac{, \vdash E : \tau' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau'}{, \vdash \mathbf{PutSP}\ E\ F : \mathbf{SP}\ \tau\ \tau'}$$

$$(\mathit{Series}) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau''}$$

$$(\mathit{Parallel}) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau'}{, \vdash E >*< F : \mathbf{SP}\ \tau\ \tau'}$$

$$(\mathit{Feedback}) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau}{, \vdash \mathbf{LoopSP}\ E : \mathbf{SP}\ \tau\ \tau}$$

$$(\mathit{Dyn}) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau'}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'}$$

Figure 4.3: Type rules for stream processors

For our remaining examples we will use stream processors over booleans, which could be used to construct circuit like networks. First, we define a boolean negation stream processor that negates values sent to it on its input stream:

$$\emptyset \vdash \mathbf{Fix}\ \mathit{neg}.(\mathbf{GetSP}\ (\lambda b.\ \mathbf{if}\ b\ \mathbf{then\ PutSP\ false}\ \mathit{neg} \\ \mathbf{else\ PutSP\ true}\ \mathit{neg})) : \mathbf{SP\ bool\ bool}.$$

Similarly, we can define a disjunction stream processor that expects to be sent two booleans on its input stream, and outputs their disjunct on its output stream:

$$\emptyset \vdash \mathbf{Fix}\ \mathit{or}.(\mathbf{GetSP}\ (\lambda b.\mathbf{GetSP}\ (\lambda c.\ \mathbf{if}\ b\ \mathbf{then\ PutSP}\ b\ \mathit{or} \\ \mathbf{else\ PutSP}\ c\ \mathit{or})) : \mathbf{SP\ bool\ bool}.$$

All these examples have been of atomic stream processors, we now consider composite stream processors. Using De-Morgan's law [Mat93] we can define a conjunction stream processor as:

$$\emptyset \vdash \mathit{neg} >==< \mathit{or} >==< \mathit{neg} : \mathbf{SP\ bool\ bool}.$$

$$(LInj) \quad \frac{, \vdash E : \tau}{, \vdash \mathbf{Left}\ E : \tau + \tau'}$$

$$(RInj) \quad \frac{, \vdash F : \tau'}{, \vdash \mathbf{Right}\ F : \tau + \tau'}$$

$$(Case) \quad \frac{, \vdash E : \tau + \tau' \quad , \vdash F : \tau \to \tau'' \quad , \vdash G : \tau' \to \tau''}{, \vdash \mathbf{Case}\ E \to F, G : \tau''}$$

$$(If) \quad \frac{, \vdash E : \mathbf{bool} \quad , \vdash F : \tau \quad , \vdash G : \tau}{, \vdash \mathbf{if}\ E\ \mathbf{then}\ F\ \mathbf{else}\ G : \tau}$$

$$(Add) \quad \frac{, \vdash E : \mathbf{num} \quad , \vdash F : \mathbf{num}}{, \vdash \mathbf{add}\ E\ F : \mathbf{num}}$$

$$(Eq) \quad \frac{, \vdash E : \mathbf{num} \quad , \vdash F : \mathbf{num}}{, \vdash \mathbf{equal}\ E\ F : \mathbf{bool}}$$

Figure 4.4: Type rules for constant functions

Finally, a simple oscillator producing alternating booleans on its output stream, starting with **true**, can be defined using the feedback combinator and the negation stream processor as follows:

$$\emptyset \vdash \mathbf{LoopSP}\ (\mathbf{PutSP}\ \mathbf{true}\ \mathit{neg}) : \mathbf{SP}\ \mathbf{bool}\ \mathbf{bool}.$$

This initially outputs the value **true**, and this is also fed into the stream processor by the feedback loop. The stream processor negates this value sent around the loop and outputs the value **false**. This is turn is fed back into the stream processor to be negated and this process continues producing alternating boolean values.

4.3 Encoding fudgets in Core Fudgets

Although the Core Fudgets language does not include any syntax for fudgets themselves, they can be encoded within the language. In this section we sketch the details of this encoding as a translation into a subset of the Haskell language. This subset of Haskell is easily mapped onto the Core Fudgets language and includes its stream processing constructs.

The main concept is to model a fudget as a stream processor. Since stream processors only have a single input and output stream, unlike fudgets, we must simulate two streams by one. This is achieved by tagging the values in the stream, to indicate which of the two virtual streams they belong to, using the datatype:

$$\textbf{data}\ \mathit{Message\ low\ high}\ =\ \mathit{Low\ low}\ \mid \mathit{High\ high}.$$

All low-level values l are represented by $Low\ l$, and similarly all high-level values h by $High\ h$. The encoding of fudgets as stream processors is thus:

$$\textbf{type}\ \mathit{F\ in\ out}\ =\ \mathit{SP}\ (\mathit{Message\ in\ Response})\ (\mathit{Message\ out\ Request}).$$

The types *Request* and *Response* are used to represent the type of values that can be sent to the window system to request a service, and with which the window system responds with, respectively.

Using this representation, fudgets can use an extended form of the *putSP* function indicating which of the two streams to use: high-level or low-level. These extended *putSP* functions simply encapsulate the value to be output inside of the appropriate constructor of the *Message* datatype:

$$\begin{array}{lcl}
\mathit{putH} & :: & \mathit{out} \to \mathit{F\ in\ out} \to \mathit{F\ in\ out} \\
\mathit{putH\ v\ k} & = & \mathit{putSP\ (High\ v)\ k}
\end{array}$$

$$\begin{array}{lcl}
\mathit{putL} & :: & \mathit{Request} \to \mathit{F\ in\ out} \to \mathit{F\ in\ out} \\
\mathit{putL\ v\ k} & = & \mathit{putSP\ (Low\ v)\ k}.
\end{array}$$

Similarly, the *getSP* function is not sufficient for defining the behaviour of fudgets. Instead, an extended form of the construct can be defined as follows:

$$\begin{array}{lcl}
\mathit{getF} & :: & \mathit{F\ in\ out} \to (\mathit{in} \to \mathit{F\ in\ out}) \to (\mathit{Request} \to \mathit{F\ in\ out}) \\
\mathit{getF\ hf\ lf} & = & \mathit{getSP}\ (\backslash m \to \textbf{case}\ m\ \textbf{of}\ \mathit{High\ v} \to \mathit{hf\ v} \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \mathit{Low\ v} \to \mathit{lf\ v}).
\end{array}$$

A term of the form, $getF\ hf\ lf$, expects to receive a value on either the low-level or high-level input stream. This value is then processed with either the *lf* or *hf* function depending upon which of the two input streams the value arrived on.

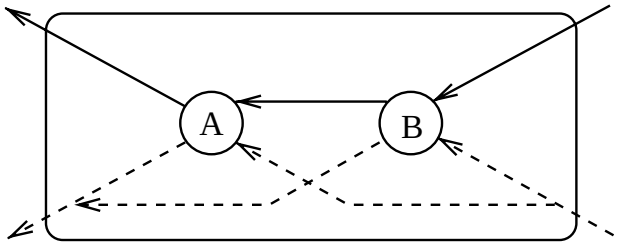


Figure 4.5: Serial composition for fudgets, $A >==<_F B$

The routing of high-level and low-level streams in the composition of fudgets is sometimes different. For example, the routing of the streams for serial composition is illustrated in Figure 4.5. This routing can be encoded with stream processors, but requires a datatype with three tags. Values sent between the two stream processors are tagged with this datatype. The third tag identifies low-level values generated from the rightmost stream processor, which must be routed around the leftmost stream processor.

The Haskell code in Figure 4.6 encodes serial composition of Fudgets using only stream processor combinators. The stream processor *postB* tags any low-level outputs from B such that they are routed around A by *idA*. Similarly, *preA* untags any low-level inputs routed around B by *idB*.

$$
\begin{array}{lcl}
>==<_F & :: & F\ mid\ out \rightarrow F\ in\ mid \rightarrow F\ in\ out \\
A>==<_F B & = & ((A >==< preA) >*< idA) \quad >==< \\
 & & ((postB >==< B) >*< idB)
\end{array}
$$

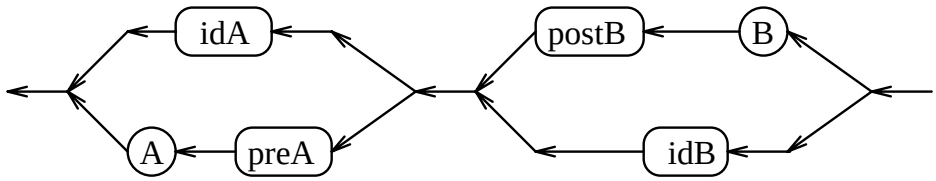


Figure 4.6: Encoding serial composition

The routing of the low-level streams for parallel composition of fudgets is identical to the high-level streams, as shown in Figure 4.7. This yields a trivial encoding of parallel composition in terms of the parallel composition of stream processors.

$$
\begin{array}{lcl}
> * <_F & :: & F\ in\ out \to F\ in\ out \to F\ in\ out \\
A> * <_F B & = & A> * <B
\end{array}
$$

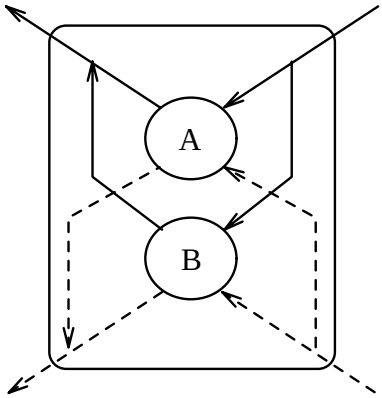


Figure 4.7: Encoding parallel composition

Feedback requires slightly different routing for its low-level streams than for its high-level streams as shown in Figure 4.8. This routing can be encoded similarly to that for serial composition, using a datatype with three tags.

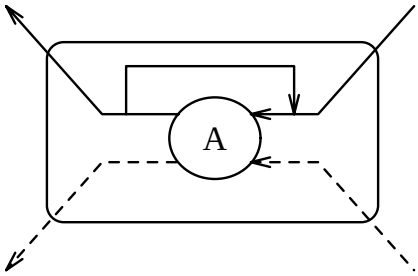


Figure 4.8: Feedback for fudgets, **LoopF** *A*

The Haskell code in Figure 4.9 encodes feedback for fudgets using only stream processor combinators. The low-level or high-level tag of any input is converted into the three tag datatype by the stream processor *init*. Next, *stripLows* ignores any low-level values routed around the feedback loop, and converts the tags of other values back into the usual low or high level tags. Lastly, *postA* tags low-level values with the third tag to ensure they are ignored by *stripLow* when fed around the loop, and *final* converts the tags back into the usual high or low level tags.

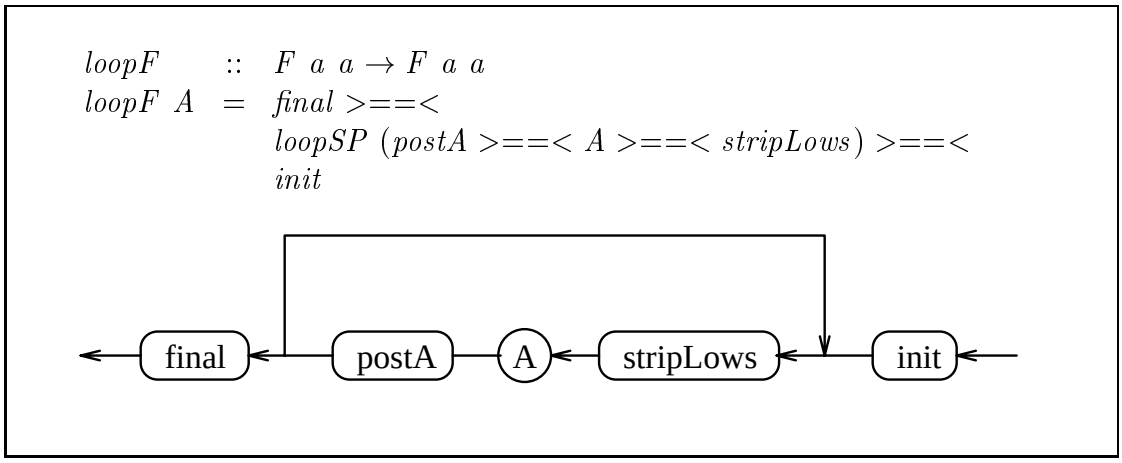


Figure 4.9: Encoding of feedback

The *dynF* combinator can be encoded using its stream processor equivalent, *dynSP*. There are no difficulties with routing the low-level streams appropriately in this case. However, the only complication is that when a new fudget is sent to a fudget constructed using *dynF* then the old fudget must be destroyed. This destruction may well involve closing a window on the screen. This can be accomplished by encoding *dynF* as a serial composition of *dynSP* and an auxiliary stream processor. This auxiliary stream processor checks for a new fudget being sent along the input stream, and if this is found forwards it but first sends a low-level value to the old fudget indicating that it should destroy any windows.

One extra detail to be considered is how a response to a low-level window request is routed back to the fudget that made the request. Hallgren and Carlsson [HC95] achieve this by pairing low-level values with a special path value. This path value indicates how a response should be routed through the network of fudgets.

Chapter 5

The π-calculus

The π-calculus is a process calculus designed for modelling mobile processes. This chapter defines the syntax, type assignment relation, and operational semantics of a minor variant of Milner, Parrow and Walker's original calculus [MPW89]. This calculus forms the basis for a formal semantics of the Core Fudgets language, which is examined in Chapter 6. This semantics describes the meaning of Core Fudget programs by translating them into functions yielding π-calculus processes.

The calculus we consider in this chapter is based on the polyadic π-calculus and includes a replication construct but has no matching construct. We introduce a typing system for this calculus, which is used in Chapter 6 to show that the formal semantics for Core Fudgets preserves types. We have chosen to use the π-calculus as the basis of a semantics for Core Fudgets because it can model higher-order processes, and has been studied in depth. Also, there are standard encodings of the λ-calculus into it, which we make use of for describing the λ-calculus components of Core Fudgets. Finally, the π-calculus has to a large degree superceded the earlier CCS process calculus as the standard process calculus.

In Section 5.1 we give an overview of the calculus, which is followed in Section 5.2 by a definition of the syntax of the calculus. The next section presents a type system for the calculus that is used to verify that the formal semantics in Chapter 6 preserves types. Finally, we conclude the chapter by describing an operational semantics for the calculus using labelled transition systems.

5.1 Overview

The π-calculus is a process calculus for describing a network of concurrent processes that may change dynamically. The most important concepts in the π-calculus are *processes* and *channels*. Processes can interact with one another by exchanging information over channels. For example, the process $c!v.P$ transmits v along the channel c and then continues as process P. Similarly the process $c?x.P$ waits for v to be transmitted along c and then continues as process P with x instantiated to v. Communication is synchronous; in the processes $c!v.P$ or $c?x.P$ the process P is prevented from executing until the communication on the channel c has occurred. The only entities that can be exchanged during communication are channels. Two processes may be executed in parallel using the parallel composition operator, $|$, which enables the processes to interact. For example, the two processes in the term $c!v.P \mid c?x.Q$ can interact as they both wish to communicate on the channel c, resulting in the term $P \mid Q[v/x]$ where $Q[v/x]$ denotes the substitution of v for x in Q. It has been shown by Sangiorgi [San92] that this is sufficient to encode more involved operations such as the communication of processes.

There are two forms of the π-calculus, monadic and polyadic. In the monadic π-calculus exactly one channel is exchanged during a communication, while in the polyadic π-calculus tuples of channels can be exchanged in an atomic communication. For example, a process of the form $c![v_1, ..., v_n]$ simultaneously transmits the tuple of channels $(v_1, ..., v_n)$ along c. The polyadic π-calculus introduces the possibility of runtime failure, when a tuple sent along a channel may not have the same length as the tuple expected by a receiving process.

Here, we consider a minor variant of the polyadic form of the π-calculus. In comparison to the π-calculus described by Milner, Parrow, and Walker [MPW89], we do not have a construct for matching but add a construct for replication as presented in Milner's later tutorial on the π-calculus [Mil91]. Replication is useful for representing processes that act as servers, and is necessary for encoding the λ-calculus and recursion. Although it is possible to encode base types such as numbers and booleans in the π-calculus, we include direct support for these types to simplify the encodings. We note that this requires extending the entities that can be exchanged during communication to include values of base types. We restrict choice to *guarded processes* that begin with an input or an output along a channel as the encoding of Core Fudgets only requires this form of choice.

5.2 Syntax

We identify two categories of syntactic objects, *values*, and *processes*. Values correspond to channels and values of base types. We presuppose an infinite set of channel variables, $x, y, ..., \in \mathcal{N}$, and the set of values, denoted $u, v, w, ... \in \mathcal{V}$, is given by the grammar:

v	::=	x	*Channel Variables*
	|	$0, 1, 2, ...$	*Numbers*
	|	$true, false$	*Booleans*

When the length of a tuple is clear from the context, or is unimportant, then we let $\vec{x}$ denote $x_1, ..., x_n$, and we will use this notation for tuples throughout the remainder of this dissertation. The set of polyadic π-calculus processes, denoted $P, Q, ..., \in \mathcal{P}$, is defined in terms of the set of guarded π-calculus processes, denoted $G, H, ..., \in \mathcal{G}$ by the grammar:

G	::=	$x?[\vec{y}].P$	*Input*
	|	$x![\vec{v}].P$	*Output*

P	::=	G	*Guarded processes*
	|	$(\nu y)P$	*Restriction*
	|	$P \mid Q$	*Parallel composition*
	|	$G + H$	*Choice*
	|	$*P$	*Replication*
	|	$\mathbf{0}$	*Nil Process*
	|	**if** v **then** P **else** Q	*Conditional*
	|	**add** $[v, w, x].P$	*Additions*
	|	**equal** $[v, w, x].P$	*Number equality*

Informally, the guarded process $x?[\vec{y}].P$ waits for a tuple of values $\vec{v}$ to be transmitted along the channel x and then continues as P where $\vec{y}$ is instantiated with the values $\vec{v}$. Similarly, the guarded process $x![\vec{v}].P$ transmits the values $\vec{v}$ along the channel x and then continues as P.

The set of processes, $\mathcal{P}$, includes all guarded processes along with the processes constructed by restriction, parallel composition, guarded choice, replication, and some constructs for base types. A restriction, $(\nu y)P$, introduces a new channel and

bind it to y in P, and we let $(\nu\vec{y})P$ stand for $(\nu y_1)(\nu y_2)...P$. Parallel composition, $P \mid Q$, allows two processes to be run concurrently, while the nil process $\mathbf{0}$ is the inactive process. We abbreviate the common form of process $x![\vec{v}].\mathbf{0}$ as $x![\vec{v}]$. We can now give some simple examples of π-calculus terms — the first is a parallel composition of two processes:

$$x![y] \mid x![z]$$

and the result is the transmission of the value y and z along the channel x. However, the concurrency of parallel composition means that we do not know which of these values is sent first. The next example illustrates synchronisation between an output process and an input process:

$$x![y].P \mid x?[z].Q$$

Here, we have a number of possibilities, first the leftmost process could transmit the value y along the channel x, followed by the rightmost process reading a value from the channel x and binding it to z. Alternatively, the rightmost process could read a value from the channel x binding it to z, followed by the leftmost process transmitting the value y along the channel x. Finally, the two processes can synchronise, and the value y is transmitted from the leftmost process to the rightmost one and is bound to z resulting in the process:

$$P \mid Q[y/z]$$

A choice, $G+H$, can either act as G or as H but not as both, with the decision being made internally to the overall process. Choice is required for encoding the dynamic stream processor construct, **DynSP**. Guarded processes are introduced to restrict choice such that the branches must begin with an input or output along a channel. The replication $*P$ represents an unbounded number of copies of P in parallel. Finally, we have three constructs for operations on base types. The conditional **if** v **then** P **else** Q behaves as P if v is *true* and Q otherwise, while the construct **add** $[v, w, x].P$ transmits the sum of the numbers v and w along the channel x and continues as P. Equality on numbers is supported by the construct **equal** $[v, w, x].P$ which compares the numbers v and w for equality. If they are equal then the value *true* is sent along the channel x, otherwise *false* is sent on x.

In the processes, $x?[\vec{y}].P$, and $(\nu\vec{y})P$, the occurrence of the channel variables $\vec{y}$ are binding occurrences, with the scope of these variables being the process P. An occurrence of a channel variable y in a process is free if it does not lie within the scope of a binding occurrence of y. The free and bound channel variables of a process are denoted by $fv(P)$ and $bv(P)$, respectively. We identify processes that differ only in the choice of bound channel variables, and use the symbol $=$ to denote syntactic equality modulo bound channel variables. We define substitution in the standard way, and write $P[\vec{v}/\vec{x}]$ for the simultaneous substitution of the values $\vec{v}$ for the channel variables $\vec{x}$ in the process P, where bound channel variables may need to be renamed to avoid the inadvertent capture of free channel variables.

5.3 Types

We make use of Turner's monomorphic type system [Tur95] for the π-calculus to ensure that channel variables are used consistently. This will also be used to check that the formal semantics in Chapter 6 preserves types of the Core Fudgets language, thus acting as a useful sanity check for the correctness of the semantics. The set of π-calculus types, denoted $\tau, \tau', ..., \in \mathcal{T}$, is given by the grammar:

τ	::=	α	*Type Variables*
	|	$\uparrow[\vec{\tau}]$	*Channel Types*
	|	**num**	*Numeric Type*
	|	**bool**	*Boolean Type*
	|	**Fix** $\alpha.\tau$	*Recursive Types*

The type of a channel that can carry tuples of type $\vec{\tau}$ is denoted as $\uparrow[\vec{\tau}]$. There are types corresponding to the two base types, booleans and natural numbers. Also, we include recursive types, **Fix** $\alpha.\tau$, which will be required in the encoding of stream processors presented in Chapter 6. This also necessitates the inclusion of type variables.

Type contexts record the types of free variables in a process, and are sequences of bindings such as $\vec{x} : \vec{\tau}$, where $\vec{x}$ is composed of distinct variables. We will use $, , \Delta$ to range over type contexts, and write $, (x)$ to denote the type assigned to the channel x by the context $, .$ A context which does not assign types to the variables $\vec{x}$ is denoted as $, _{\vec{x}}$.

The type rules for values are shown in Figure 5.1, where a typing judgement $, \vdash v : \tau$ asserts the value v has the type τ under the context $,$. Processes have no result value and so we define typing judgements for them of the form $, \vdash P$ by the least relation satisfying the rules in Figure 5.2. Intuitively, $, \vdash P$ asserts that the process P uses its free channel variables consistently with the context $,$.

Recursive types are treated by allowing implicit folding and unfolding. Type equality, $\simeq$, is defined to capture the isomorphism between a recursive type and any unfolding of this type. Turner defines $\simeq$ to equate two types if the same observations can be made of both types. An observation of a type indicates if the type is a channel type or a base type. A restriction of this approach is that the recursive types must be contractive in the recursive argument – in the type **Fix** $\alpha.\tau$ the variable α must occur inside at least one channel type constructor. This ensures that there is a unique observation for every type. We adopt this approach for handling recursive types and use $\simeq$ as equality on types.

$$(True) \quad , \vdash \mathit{true} : \mathbf{bool} \qquad (False) \quad , \vdash \mathit{false} : \mathbf{bool}$$

$$(Var) \quad , , x : \tau \vdash x : \tau \qquad (Num) \quad \frac{n \in \{0, 1, ...\}}{, \vdash n : \mathbf{num}}$$

Figure 5.1: Type rules for values

We can add a new binding for a variable x without invalidating the typing of a process P as long as $x \notin FV(P)$. This property will be used to show that the π-calculus semantics of Chapter 6 preserves Core Fudget types:

Lemma 5.1 (Weakening) *if* $, \vdash P$ *and* $x \notin fv(P)$ *then* $, , x : \tau \vdash P$.

Proof. *A straightforward induction on the structure of* P. ∎

The standard substitution lemma can be reformulated for our variant of the π-calculus as follows, and is useful in proving subject reduction for the operational semantics of the π-calculus given in the next section.

Lemma 5.2 (Substitution) *Given* $, , \vec{x} : \vec{\tau} \vdash P$ *and a set of values,* $, \vdash \vec{v} : \vec{\tau}$ *then* $, \vdash P[\vec{v}/\vec{x}]$.

Proof. *A simple induction on the structure of* P. ∎

$$(\mathit{Input}) \quad \frac{, (c) \simeq\uparrow [\vec{\tau}] \quad , , \vec{x} : \vec{\tau} \vdash P}{, \vdash c?[\vec{x}].P}$$

$$(\mathit{Output}) \quad \frac{, \vdash \vec{v} : \vec{\tau} \quad , (c) \simeq\uparrow [\vec{\tau}] \quad , \vdash P}{, \vdash c![\vec{v}].P}$$

$$(\mathit{Res}) \quad \frac{, , x : \tau \vdash P \quad \tau \simeq\uparrow [\tau']}{, \vdash (\nu x)P}$$

$$(\mathit{Prl}) \quad \frac{, \vdash Q \quad , \vdash P}{, \vdash P \mid Q}$$

$$(\mathit{Choice}) \quad \frac{, \vdash G \quad , \vdash H}{, \vdash G + H}$$

$$(\mathit{Repl}) \quad \frac{, \vdash P}{, \vdash *P}$$

$$(\mathit{Nil}) \quad , \vdash \mathbf{0}$$

$$(\mathit{Cond}) \quad \frac{, \vdash b : \mathbf{bool} \quad , \vdash P \quad , \vdash Q}{, \vdash \mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q}$$

$$(\mathit{Sum}) \quad \frac{, \vdash v : \mathbf{num} \quad , \vdash w : \mathbf{num} \quad , (x) \simeq\uparrow [\mathbf{num}] \quad , \vdash P}{, \vdash \mathbf{add}\ [v, w, x].P}$$

$$(\mathit{Equal}) \quad \frac{, \vdash v : \mathbf{num} \quad , \vdash w : \mathbf{num} \quad , (x) \simeq\uparrow [\mathbf{bool}] \quad , \vdash P}{, \vdash \mathbf{equal}\ [v, w, x].P}$$

Figure 5.2: Type rules for processes

An example of a typing derivation is shown in Figure 5.3 for the process $(\nu x, y)(x?[z].\mathbf{0} \mid y![z])$. This uses the rules *Nil*, *Input*, *Output*, *Prl*, and *Res*. We can see from this derivation that the type of the channel z determines the type of the channels x, and y.

$$\dfrac{\dfrac{\dfrac{, , x :\uparrow [\tau], y :\uparrow\uparrow [\tau], z : \tau \vdash \mathbf{0}}{, , x :\uparrow [\tau], y :\uparrow\uparrow [\tau] \vdash x?[z].\mathbf{0}} \quad \dfrac{, , x :\uparrow [\tau], y :\uparrow\uparrow [\tau] \vdash \mathbf{0}}{, , x :\uparrow [\tau], y :\uparrow\uparrow [\tau] \vdash y![x].\mathbf{0}}}{, , x :\uparrow [\tau], y :\uparrow\uparrow [\tau] \vdash x?[z].\mathbf{0} \mid y![x].\mathbf{0}}}{, \vdash (\nu x, y)(x?[z].\mathbf{0} \mid y![x].\mathbf{0})}$$

Figure 5.3: An example typing derivation

This example also illustrates a common shorthand notation we will use for channels that can carry channels. Instead of writing $\uparrow [\uparrow [\tau]]$ we will denote this type by $\uparrow\uparrow [\tau]$ and this shorthand extends naturally to multiply nested channel types.

5.4 Semantics

We present the semantics of the π-calculus in terms of a labelled transition relation. An alternative approach is to use a reduction relation and a structural congruence relation in the style of Milner [Mil90, Mil91]. We do not take this approach here, as the labelled transition relation simplifies the definition of equivalences on π-calculus terms that we present in Chapter 7.

A labelled transition system is a triple, $(S, A, \xrightarrow{\alpha})$, of a set of states S, a set A of actions used to label transitions, and a transition relation $\xrightarrow{\alpha} \subseteq S \times S$ for each $\alpha \in A$. Following Sewell [Sew97], we extend this definition to include a typing context, and define the transition relation as $, \vdash P \xrightarrow{\alpha} Q$, meaning that $, \vdash P$ and P can evolve into Q by performing the action α. The typing context is required to ensure that only values of the correct type are communicated along channels. We will consider three possible actions:

- τ, a *silent* action, which represents an interaction between two processes that are ready to input and output, respectively, on the same channel.

- $x?\langle\vec{v}\rangle$, an *input* action, which represents the possibility of a process receiving input along x, where the values received are $\vec{v}$.
- $(\nu\vec{y'})x!\langle\vec{v}\rangle$, an output action, which represents the possibility of a process transmitting $\vec{v}$ along x. The $\vec{y'}$ represent private channel variables that are emitted from the process and are thus carried out from their current scope. It will always be the case that $\vec{y'} \subseteq \vec{v} - x$.

Different brackets are used in input prefixes $x?[y]$ as opposed to input actions, $x?\langle y\rangle$. This is to make explicit that in an input prefix, $x?[\vec{y}]$, the channel variables $\vec{y}$ are binders waiting to be instantiated whereas in an input action, $x?\langle\vec{y}\rangle$ they represent values with which the binders have been instantiated. In a similar way the variables $\vec{y}$ in an output prefix, $x![\vec{y}]$, are considered as values in an output action, $x!\langle\vec{y}\rangle$.

We will use α to range over actions, and denote the channel variables of an action by $v(\alpha)$. The free and bound variables of actions, $fv(\alpha)$ and $bv(\alpha)$, are defined in Figure 5.4.

α	$fv(\alpha)$	$bv(\alpha)$
$x?\langle\vec{y}\rangle$	$\{x\} \cup \vec{y}$	$\emptyset$
$(\nu\vec{y'})x!\langle\vec{y}\rangle$	$\{x\} \cup \vec{y} - \vec{y'}$	$\vec{y'}$
τ	$\emptyset$	$\emptyset$

Figure 5.4: Free and bound variables

The transition relation is defined by the rules in Figure 5.5, with symmetric versions of (Par), (Sum), and (Com) omitted. Processes differing in their bound channels have the same transitions modulo α-renaming. This is an *early* semantics [MPW89] as bound channels are instantiated in the (In) rule rather than the (Com) rule. The hypotheses for the (In) rule ensure that the values received have types corresponding to the type of the channel upon which they are received.

The transition, , $\vdash P \xrightarrow{x?\langle z\rangle} P'$, illustrates the side condition on the (Par) rule which avoids instantiating z in Q when a channel is actually received. The (Com) rule's side condition stops free occurrences in Q' of the private channels that are emitted from P being incorrectly bound. The side condition on the (Res) rule stops any channel variables received by P being incorrectly bound.

$$(Add) \quad , \ _{l,m} \vdash \mathbf{add}\ [l,m,r].P \xrightarrow{\tau} r![l+m].P$$

$$(Equal_T) \quad , \ _{l,m} \vdash \mathbf{equal}\ [l,m,r].P \to \tau r![true].P \qquad l = m$$

$$(Equal_F) \quad , \ _{l,m} \vdash \mathbf{equal}\ [l,m,r].P \to \tau r![false].P \qquad l \neq m$$

$$(If_T) \quad , \vdash \mathbf{if}\ true\ \mathbf{then}\ P\ \mathbf{else}\ Q \xrightarrow{\tau} P$$

$$(If_F) \quad , \vdash \mathbf{if}\ false\ \mathbf{then}\ P\ \mathbf{else}\ Q \xrightarrow{\tau} Q$$

$$(Out) \quad , \vdash x![\vec{v}].P \xrightarrow{x!\langle\vec{v}\rangle} P$$

$$(In) \quad \frac{, \vdash \vec{v} : \vec{\tau} \quad , (x) \simeq\uparrow [\vec{\tau}]}{, \vdash x?[\vec{y}].P \xrightarrow{x?\langle\vec{v}\rangle} P[\vec{v}/\vec{y}]}$$

$$(Sum) \quad \frac{, \vdash G \xrightarrow{\alpha} G'}{, \vdash G + H \xrightarrow{\alpha} G'}$$

$$(Par) \quad \frac{, \vdash P \xrightarrow{\alpha} P'}{, \vdash P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad bv(\alpha) \cap fv(Q) = \emptyset$$

$$(Com) \quad \frac{, \vdash P \xrightarrow{(\nu\vec{y'})x!\langle\vec{v}\rangle} P' \quad , \vdash Q \xrightarrow{x?\langle\vec{v}\rangle} Q'}{, \vdash P \mid Q \xrightarrow{\tau} (\nu\vec{y'})(P' \mid Q')} \qquad \vec{y'} \cap fv(Q) = \emptyset$$

$$(Res) \quad \frac{, , x : \tau \vdash P \xrightarrow{\alpha} P' \quad \tau \simeq\uparrow [\tau']}{, \vdash (\nu x)P \xrightarrow{\alpha} (\nu x)P'} \qquad x \notin v(\alpha)$$

$$(Open) \quad \frac{, , x : \tau \vdash P \xrightarrow{(\nu\vec{y'})z!\langle\vec{v}\rangle} P' \quad \tau \simeq\uparrow [\tau']}{, \vdash (\nu x)P \xrightarrow{(\nu x,\vec{y'})z!\langle\vec{v}\rangle} P'} \qquad x \neq z, x \in \vec{v} - \vec{y'}$$

$$(Rep) \quad \frac{, \vdash P \mid *P \xrightarrow{\alpha} P'}{, \vdash *P \xrightarrow{\alpha} P'}$$

Figure 5.5: Semantics for the π-calculus

Lemma 5.3 (Subject reduction) *if* , $\vdash P \xrightarrow{\alpha} P'$ *then* , $\vdash P'$

Proof. *By induction on the structure of the derivation* , $\vdash P \xrightarrow{\alpha} P'$. *We examine the cases for input and output processes:*

Case , $\vdash x![\vec{v}].P' \xrightarrow{x!\langle\vec{v}\rangle} P'$. *Since* , $\vdash x![\vec{v}].P'$, *which must be derived from the (Output) rule, one of whose hypotheses is* , $\vdash P'$, *the required result.*

Case , $\vdash x?[\vec{y}].P' \xrightarrow{x?\langle\vec{v}\rangle} P'[\vec{v}/\vec{y}]$. *From the (In) rule it must be that* , $\vdash v_i : \tau_i$ *for* $i \in \{1, ..., n\}$. *The derivation* , $\vdash x?[\vec{y}].P'$ *must result from an application of the (Input) rule, which has the hypothesis* , , $y_1 : \tau_1, ..., y_n : \tau_n \vdash P'$. *By using Lemma 5.2 we can conclude that* , $\vdash P'[\vec{v}/\vec{y}]$ *as desired.*

The other cases follow in a similar manner. ∎

We conclude this section by illustrating the semantics with some examples. Firstly, we consider the base types and the following process indicates how three numbers can be summed:

$$\begin{array}{lll} & \textbf{add}\ [v_1, v_2, x] \mid x?[w].\textbf{add}\ [v_3, w, y] & \\ \xrightarrow{\tau} & x![v_1 + v_2] \mid x?[w].\textbf{add}\ [v_3, w, y] & (Add) \\ \xrightarrow{\tau} & (\textbf{add}\ [v_3, w, y])[v_1 + v_2/w] & (Com) \\ = & \textbf{add}\ [v_3, v_1 + v_2, y] & \\ \xrightarrow{\tau} & y![v_3 + (v_1 + v_2)] & (Add) \end{array}$$

The process is a parallel composition of two processes, the first sums the values v_1 and v_2 transmitting the result along the channel x. The second process awaits this resulting value to be sent on the channel x and then continues as the process **add** $[v_3, w, y]$ with the result of the first addition substituted for w. This will add the value v_3 to the sum and the overall result is sent along the channel y. This illustrates why transition rules are not required to reduce the arguments to the addition construct as the arguments can only be natural numbers. However, this doesn't limit the use of the construct as variables can be used in place of actual numbers, as in the above example. Note that + in this example denotes the meta-construct of addition and not the choice operator of the π-calculus itself.

The following process, which communicates a local channel beyond its original scope, illustrates why a restriction is required as part of an output action:

$$y?[z].P \mid (\nu x)(y![x].Q) \quad \xrightarrow{\tau} \quad (\nu x)(P[x/z] \mid Q) \quad (Com).$$

After the communication occurs the channel x can be used in the process P, and thus the scope of x has expanded beyond the process $y![x].Q$. This is called scope extrusion and the semantics handles this by including any private channel variables that are transmitted along a channel during an output action in the action itself.

5.4.1 Recursive process definitions

Most presentations of the π-calculus include a replication operator, such as the one described in this chapter, to enable processes to have infinite behaviour. The first descriptions of the π-calculus do not include a replication operator, but instead have a facility to define recursive processes. The replication operator neatly replaces this complicated mechanism, but we will find it useful in the rest of this thesis to make use of recursively defined processes:

Definition 5.1 (Recursive process definition) *A recursive process definition is specified by a unique defining equation:*

$$A(\vec{x}) = P$$

where the channel variables $\vec{x}$ are distinct and are the only variables which occur free in the process P. An instantiation of such a process definition, $A(\vec{y})$, behaves like the process $P[\vec{y}/\vec{x}]$.

These process definitions can be entirely translated into the π-calculus described in this chapter. For example, the process P in the context of the recursive process definition $A(x, y) = x?[z].y![z].A(x, y)$ is translated into the process $(\nu A)(*A?[x, y].x?[z].y![z].A![x, y] \mid P)$. A new channel variable is created for the process definition, and a replicated input process waits to receive the arguments. In the body of the definition any calls to process definitions are translated by sending the actual arguments to the process definition being invoked along the channel corresponding to the process definition. The process P runs in parallel with the replicated input process allowing it to call the definition A by sending the actual arguments along the channel A.

Chapter 6

Denotational semantics of Core Fudgets

In this chapter, we describe a formal semantics for the Core Fudgets language as a translation into the π-calculus of the previous chapter. The semantics is denotational and we give a brief overview of what this means in Section 6.1. Next we explain the main principle of the translation from Core Fudgets to the π-calculus, $[\![\_]\!]$, and describe how literals can be encoded in the π-calculus. This is followed in Section 6.4 by a review of the semantics of the λ-calculus as a translation into the π-calculus. This suggests a natural method for encoding stream processors and their combinators, which we describe in Section 6.5. The chapter concludes with a result showing that the translation into the π-calculus preserves the type structure of the Core Fudgets language.

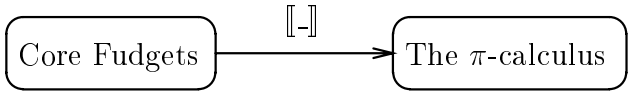


Figure 6.1: Denotational semantics for Core Fudgets

6.1 Denotational semantics

The semantics we describe here is *denotational* [Sch86], and maps terms directly to meanings, called denotations. Usually, the denotation is a mathematical value, such as a number or a function. There is no explicit concept of a machine or computation sequence. An important property of a denotational semantics is *compositionality*. This corresponds to the denotation of expressions being defined purely in terms of the denotations of their subexpressions. This property is important as it means that equational reasoning can be used to manipulate programs simply by replacing one term with another whose semantic denotation is the same.

6.2 A π-calculus semantics

We base the semantics on the π-calculus semantics of the λ-calculus. Milner [Mil90] first presented such an encoding, but we will make use of Ostheimer and Davie's encoding [OD93] because it is more readily amenable to typing using Turner's type system for the π-calculus [Tur95].

The translation maps Core Fudget terms, E, to encodings $[\![E]\!]$. These encodings are functions from π-calculus channels to π-calculus processes. Applying this function to a π-calculus channel, o, results in a π-calculus process. The parameter o specifies the location where the result of evaluating the term will be deposited. This is the opposite to Milner's original encoding, where the channel o represents the location where the term E receives its arguments. Ostheimer and Davie [OD93] give a concise explanation of this difference: "Milner's scheme has no output—this scheme has no input". The concept of a *result channel*, such as o, will be used extensively throughout the semantics.

The semantics for the Core Fudgets language is given by a semantic function from terms to functions taking π-calculus channels to π-calculus processes:

$$[\![\_]\!] : Expr \to (\mathcal{N} \to \mathcal{P}).$$

The function corresponding to a Core Fudgets term expects to receive a π-calculus channel which indicates the location where the evaluated term should be deposited. We overload the syntax and use $[\![\_]\!]$ to denote the translation of Core Fudgets types and also the translation of type contexts to π-calculus type contexts.

6.3 A π-calculus semantics of literals

Milner [MPW89] illustrated how datatypes can be translated into the π-calculus using standard Church-style encodings. We have chosen not to use these encodings for the base types, but to support these directly in our variant of the π-calculus. This simplifies the encodings of Core Fudget programs greatly.

We first consider the booleans, and their translation into the π-calculus. The two literals **true** and **false**, the conditional construct and the translation of the boolean type, **bool**, can be encoded as shown in Figure 6.2.

$$
\begin{array}{lcl}
[\![\textbf{bool}]\!] & = & \textbf{bool} \\
[\![\textbf{true}]\!]\,o & = & o![true] \\
[\![\textbf{false}]\!]\,o & = & o![false] \\
[\![\textbf{if } E \textbf{ then } F \textbf{ else } G]\!]\,o & = & (\nu r)([\![E]\!]r \mid r?[b].\textbf{if } b \textbf{ then } [\![F]\!]o \textbf{ else } [\![G]\!]o)
\end{array}
$$

Figure 6.2: An encoding of booleans

The encoding transmits the corresponding boolean values on the result channel o. The conditional construct is a little more involved. First, it evaluates the boolean expression and retrieves the boolean from the channel r. This is then examined and either the expression F or the expression G is evaluated with the result being deposited on the result channel o. The type translation of booleans simply maps the boolean type of the Core Fudgets language to the built-in boolean base type in our variant of the π-calculus.

Natural numbers are encoded in a similar style to booleans. We make use of the natural numbers explicit in our variant of the π-calculus, as shown in Figure 6.3. The extension of the π-calculus with natural numbers has been chosen to correspond to the same interface that would be obtained if we actually encoded numbers as Church-numerals.

We choose to model sum types using a church-style encoding, but first must introduce the concept of *environment entries*. Variables in the Core Fudgets language are represented by π-calculus channels, and bindings of terms to variables are represented by environment entries.

$$
\begin{array}{lcl}
[\![\mathbf{num}]\!] & = & \mathbf{num} \\
[\![\mathbf{n}]\!]o & = & o![n] \\
[\![\mathbf{add}\ i\ j]\!]o & = & (\nu n, m)([\![i]\!]n \mid [\![j]\!]m \mid n?[x].m?[y].\mathbf{add}[x, y, o]) \\
[\![\mathbf{equal}\ i\ j]\!]o & = & (\nu n, m)([\![i]\!]n \mid [\![j]\!]m \mid n?[x].m?[y].\mathbf{equal}[x, y, o)
\end{array}
$$

Figure 6.3: An encoding of natural numbers

Definition 6.1 (Environment entry) *An* environment entry*, representing a binding of a variable x to a term M is modelled in the π-calculus as follows:*

$$[\![x := M]\!] \quad = \quad *x?[c].[\![M]\!]c.$$

This models a variable as a replicated input process that evaluates the term corresponding to the variable, and deposits it on a result channel. This concept can be used to translate binary sums as shown in Figure 6.4. A value of a sum type is encoded as a channel variable s, and a replicated input process. The variable s can be used to interrogate the sum's value by sending two channels l and r along it. If the sum's value is a left injection then it is sent on the l channel, otherwise it is sent on the r channel. The sum's actual value is represented using an environment entry, and the channel variable communicated over l or r can be used to trigger the evaluation of it. When the actual value is required a result channel is sent on the trigger channel, synchronising with the replicated input in the environment entry which evaluates the actual value and deposits it on the result channel.

The encoding of the case construct initially evaluates the value of the sum type. This is interrogated by sending it two channels, l and r, and spawning two parallel processes listening for a response on these channels. The channel received on either l or r corresponds to the sum's value and the appropriate function, F or G, is applied to it – and the processes performing this application are explained in the next section. The application of the terms F and G to the variable x will at some point demand the value of the variable x which is supplied by the environment entry corresponding to the encoding of the sum's value.

The type translation indicates the type of the channel variable s used in the encodings of **Left** E and **Right** E. A sum type corresponds to a channel on which two other channels are transmitted. These two channels carry values that can be used to trigger the evaluation of terms.

$$
\begin{array}{lcl}
[\![\tau + \tau']\!] & = & \uparrow[\uparrow\uparrow\uparrow [\![\tau]\!], \uparrow\uparrow\uparrow [\![\tau']\!]] \\
[\![\textbf{Left } E]\!]o & = & (\nu s)(o![s] \mid *s?[l,r].(\nu x)(l![x] \mid [\![x := E]\!])) \\
[\![\textbf{Right } E]\!]o & = & (\nu s)(o![s] \mid *s?[l,r].(\nu x)(r![x] \mid [\![x := F]\!])) \\
[\![\textbf{Case } E \to F, G]\!]o & = & (\nu t)([\![E]\!]t \mid t?[s].(\nu l, r)(s![l,r] \mid \\
 & & \quad l?[x].(\nu c)([\![F]\!]c \mid c?[f].f![x,o]) \mid \\
 & & \quad r?[x].(\nu c)([\![G]\!]c \mid c?[f].f![x,o])))
\end{array}
$$

Figure 6.4: An encoding of binary sums

6.4 A π-calculus semantics of the λ-calculus

We make use of Ostheimer and Davie's [OD93] encoding of the λ-calculus in which λ-calculus variables are simulated by corresponding π-calculus channel variables. Whenever the value of the variable is demanded, a result channel is sent along this channel variable indicating where the value of the variable should be deposited. Functions are translated to processes that expect to receive two channels — the argument to the function, and the location to deposit the result of applying the function. Application can be translated in a number of ways corresponding to the calling-convention required. Here, we will only consider call-by-name, but it is straightforward to encode call-by-value and call-by-need. Recursive definitions are translated by using the replication operator of the π-calculus to create a server modelling the recursively defined variable.

The type and term translations of the constructs of the λ-calculus are shown in Figure 6.5. Note that the type translation illustrates the representation of a function as a channel on which which the argument to the function will be received along with a result channel. The type of the argument allows different calling conventions because the argument is a trigger that may cause the evaluation of the argument or may simply retrieve the already evaluated argument.

$$
\begin{array}{lcl}
[\![\tau \to \tau']\!] & = & \uparrow [\uparrow\uparrow [\![\tau]\!], \uparrow [\![\tau']\!]] \\
[\![x]\!]o & = & x![o] \\
[\![\lambda x.E]\!]o & = & (\nu f)(o![f] \mid *f?[x,b].[\![E]\!]b) \\
[\![E\ F]\!]o & = & (\nu c)([\![E]\!]c \mid c?[f].(\nu x)(f![x,o] \mid [\![x := F]\!])) \\
[\![\mathbf{Fix}\ x.E]\!]o & = & (\nu x)([\![x := E]\!] \mid [\![E]\!]o)
\end{array}
$$

Figure 6.5: An encoding of the λ-calculus

The encoding of an application $E\ F$ starts by evaluating E. A function results from this which is deposited on a channel c. The term F is bound to an argument x using an environment entry, and the evaluated term E is then applied to this. This encoding models call-by-name semantics as the argument x will only be evaluated when the function E demands its value. Also if its value is demanded multiple times then it is re-evaluated each time.

Modelling other calling-conventions is straightforward. Modifying the encoding in Figure 6.5 to correspond to call-by-need semantics just involves changing the environment entry used in the encoding of application to cache the evaluated term. Call-by-value semantics can be simulated by forcing the arguments to applications to be evaluated before actually applying the function by sending the channel corresponding to the argument to the function along with a result channel. The specific order in which the function and corresponding argument are evaluated can be controlled by synchronisation on the channels where the results are deposited.

Recursive definitions are encoded by evaluating the body of the definition in parallel with an environment entry that binds the recursive variable to the body. Every time the value of the recursive variable is demanded a new process will be spawned by the replicated input process corresponding to the environment entry which evaluates the recursive call.

The type translation can be used to induce a translation on contexts. We do not merely apply the encoding of types in a pointwise fashion because the free variables range over channels that can be used to trigger the evaluation of the variable. Instead an extra channel is required, representing this trigger, as shown in Figure 6.6.

$$[\![x_1 : \tau_1, ..., x_n : \tau_n]\!] \quad = \quad x_1 :\uparrow\uparrow [\![\tau_1]\!], ..., x_n :\uparrow\uparrow [\![\tau_n]\!]$$

Figure 6.6: An encoding of type contexts

6.4.1 An Example

As an example of the λ-calculus encoding, we illustrate the behaviour of the identity term. In this example, we make use of the transition relation defining the π-calculus semantics, and also a relation, $\equiv$, defined by the following equations:

$$\begin{aligned} (\nu x)P &\equiv P \quad \textit{if } x \notin fv(P) \\ (\nu x)(*x?[y] \mid P) &\equiv P \quad \textit{if } x \notin fv(P). \end{aligned}$$

This relation allows us to eliminate unused private variables and replicated input processes that can never receive any input. The following sequence of equivalences and transitions illustrates the behaviour of the identity term:

$$\begin{aligned} & [\![(\lambda x.x)E]\!]o \\ = \quad & (\nu c)([\![\lambda x.x]\!]c \mid c?[f].(\nu x)(f![x, o] \mid [\![x := E]\!])) \\ = \quad & (\nu c)((\nu f)(c![f] \mid *f?[x, b].[\![x]\!]b) \mid c?[f].(\nu x)(f![x, o] \mid [\![x := E]\!])) \\ \xrightarrow{\tau} \quad & (\nu c, f)(*f?[x, b].[\![x]\!]b \mid (\nu x)(f![x, o] \mid [\![x := E]\!])). \end{aligned}$$

The identity function is already a value here, and so no evaluation is required and the channel f is immediately communicated along c. The next transition corresponds to the actual application of the function to the argument:

$$\xrightarrow{\tau} \quad (\nu c, f)(*f?[x, b].[\![x]\!]b \mid (\nu x)([\![x]\!]o \mid [\![x := E]\!])).$$

The replicated input process can be eliminated now as there is no way in which it can receive input:

$$\begin{aligned} \equiv \quad & (\nu x)([\![x]\!]o \mid [\![x := E]\!]) \\ = \quad & (\nu x)(x![o] \mid *x?[c].[\![E]\!]c). \end{aligned}$$

Finally, we evaluate the body of the function, which forces us to lookup the definition of the variable x in the environment entry, yielding the term E:

$$\begin{aligned} \xrightarrow{\tau} \quad & (\nu x)(*x?[c].[\![E]\!]c \mid [\![E]\!]o) \\ \equiv \quad & [\![E]\!]o. \end{aligned}$$

The resulting π-calculus term $[\![E]\!]o$ corresponds exactly to the encoding of the original term if we had β-reduced the application of the identity function. In the next chapter we will show how we can formally capture the equivalence of these terms using the concept of bisimulation.

6.5 A π-calculus semantics of stream processors

The encoding of the λ-calculus leads to a natural extension for representing stream processors. A stream processor is represented similarly to a function, as a channel on which two channels are expected to be received. These two channels correspond to the input and output stream to use for the stream processor. In general the translation of an arbitrary stream processor, S, will be of the form:

$$[\![S]\!]o \quad = \quad (\nu s)(o![s] \mid s?[in, out].P).$$

The process P may transmit the *in* and *out* channels along another channel that represents a stream processor. This corresponds to passing the input and output streams onto a continuation stream processor and is used in the translation of terms involving **GetSP** and **PutSP** because both of these take continuation arguments. Note that the input from the channel s is *not* replicated as the input and output streams will only be sent once. To simplify the presentation of the encoding, we define some notation:

Definition 6.2 (Stream processor encoding) *We define $\lceil P \rceil$ to be a π-calculus process corresponding to a generic stream processor whose input and output streams are located by the channels in and out. The behaviour of this stream processor is specified by the π-calculus process P:*

$$\lceil P \rceil \quad = \quad (\nu s)(o![s] \mid s?[in, out].P).$$

Definition 6.3 (Stream processor instantiation) *We define $a\langle in, out\rangle$ to correspond to the instantiation of the stream processor located by the channel a with the input and output streams in and out:*

$$a\langle in, out\rangle \quad = \quad a?[s].s![in, out].$$

Streams themselves will be modelled by unbounded buffers, thus considering the wires between stream processors as buffers. This ensures that no constraints are made on the possible order in which the actions of individual stream processors are undertaken.

6.5.1 Lists

Before considering the actual encoding of atomic and composite stream processors we describe how lists can be encoded in the π-calculus. These will be used in the next section to model unbounded buffers which are essential in the encoding of stream processors. This is because we will consider the wires between stream processors as unbounded buffers. Two parameterised processes are introduced corresponding to the basic operations for constructing lists, along with a type synonym, and are presented in Figure 6.7.

$$
\begin{array}{lcl}
\mathit{List}\ \tau & = & \mathbf{Fix}\ L.\uparrow[\uparrow[],\uparrow[\tau, L]] \\
\mathit{Nil}(r) & = & (\nu l)(r![l] \mid *l?[n, c].n![]) \\
\mathit{Cons}(h, t, r) & = & (\nu l)(r![l] \mid *l?[n, c].c![h, t])
\end{array}
$$

Figure 6.7: An encoding of lists

The process $\mathit{Nil}(r)$ creates a new channel corresponding to an empty list and deposits it on the channel r. The list can be interrogated by sending two channels along it to test if the list is empty, or composed of an element followed by a list. Similarly, the process $\mathit{Cons}(h, t, r)$ creates a new channel corresponding to a list which has h as its head element, and is followed by the list t. The type synonym $\mathit{List}\ \tau$ denotes the type of lists with elements of type τ. This synonym makes use of recursive types as the tail of a nonempty list is another list of the same type.

It will be useful to have a reversing operation on lists and this is defined as the *Rev* process shown in Figure 6.8. $\mathit{Rev}(l, r)$ reverses the list l and deposits the reversed list on the channel r. This makes use of an auxiliary process, $\mathit{Rev}'(l, a, r)$ that uses an accumulating parameter. $\mathit{Rev}'(l, a, r)$ reverses the list l and attaches it to the front of the list a, with the final list being deposited on the result channel r. The list a acts as an accumulating parameter. The list l is interrogated and if

empty then the result of the reversing operation is the list a. However, if the list is nonempty then a recursive call to the reversing operation is made with the tail, and a list consisting of the head followed by the accumulating parameter, with the result channel unchanged.

$$
\begin{array}{lcl}
Rev'(l,a,r) & = & (\nu n,c)(l![n,c] \mid \\
 & & \quad n?[].r![a] \mid \\
 & & \quad c?[h,t].(\nu m)(Cons(h,a,m) \mid m?[u].Rev'(t,u,r)) \\
Rev(l,r) & = & (\nu n)(Nil(n) \mid n?[m].Rev'(l,m,r))
\end{array}
$$

Figure 6.8: Reversing lists

The following lemma is useful for showing that the encoding into the π-calculus preserves types of the Core Fudgets system.

Lemma 6.1 (List types) *The type of the list operations is given by:*

- $r :\uparrow [List\ \tau] \vdash Nil(r)$;
- $h : [\![\tau]\!], t : List\ \tau, r :\uparrow [List\ \tau] \vdash Cons(h,t,r)$;
- $l : List\ \tau, r :\uparrow [List\ \tau] \vdash Rev(l,r)$.

Proof. *The proof of each part is a straightforward typing derivation. We illustrate the first part only, in Figure 6.9, with the other parts following in a similar manner.*

$$
\dfrac{\dfrac{r :\uparrow [List\ \tau], l : List\ \tau \vdash r![l] \qquad \dfrac{\dfrac{n :\uparrow [], c :\uparrow [[\![\tau]\!], List\ \tau] \vdash n![]}{l : List\ \tau \vdash l?[n,c].n![]}}{l : List\ \tau \vdash *l?[n,c].n![]}}{r :\uparrow [List\ \tau], l : List\ \tau \vdash r![l] \mid *l?[n,c].n![]}}{r :\uparrow [List\ \tau] \vdash (\nu l)(r![l] \mid *l?[n,c].n![])}
$$

Figure 6.9: A typing derivation for Nil

■

6.5.2 Unbounded buffers

We can now describe how to encode unbounded buffers by using lists. Three new processes are introduced corresponding to the construction and destruction operators for the buffers. They are defined in Figure 6.10 along with a type synonym for buffers.

$$
\begin{array}{lcl}
\mathit{Buffer}\ \tau & = & \uparrow[\mathit{List}\ \tau] \\
\\
\mathit{Empty}(r) & = & (\nu b)(r![b] \mid \mathit{Nil}(b)) \\
\mathit{Insert}(x,b,a) & = & b?[l].(\mathit{Cons}(x,l,b) \mid a![]) \\
\mathit{Remove}(b,r) & = & b?[l].(\nu s)(\mathit{Rev}(l,s) \mid \\
 & & \quad (\nu n,c)(s?[rl].rl![n,c] \mid \\
 & & \quad\quad (n?[].b![l] \mid \mathit{Remove}(b,r)) \mid \\
 & & \quad\quad c?[h,t].r![h].\mathit{Rev}(t,b)))
\end{array}
$$

Figure 6.10: Buffers

The process $\mathit{Empty}(r)$ creates a new unbounded buffer which is initially empty. The buffer is modelled as a list which is created using the $\mathit{Nil}(b)$ process. Elements are inserted into the buffer by the process $\mathit{Insert}(x,b,a)$, which adds the element represented by x to the buffer b. This first obtains exclusive access to the list by retrieving the channel corresponding to the list modelling the buffer. Once this is done then a new list is constructed using the $\mathit{Cons}(x,l,b)$ process, with the resulting list deposited on the channel representing the buffer, b. The channel a acts as an acknowledgement to the process requesting the insertion, indicating when the operation has actually taken place. Removal of elements is handled by the $\mathit{Remove}(b,r)$ process, which first retrieves the list corresponding to the buffer ensuring exclusive access. Next, this list is reversed in preparation for removing the last element, giving a first-in first-out semantics to the buffer. The reversed list is interrogated and if empty then the original list representing the buffer is output on b allowing other processes to use the list. In parallel with this, the original request to remove an element is repeated since the attempt to remove an element failed due to the buffer being empty. If the reversed list is non-empty then the head element of the list is sent to the result channel r, while the tail is reversed again to obtain the new list modelling the resulting buffer which is deposited on

the channel b allowing other processes to use the modified buffer. The types of the buffer operations are given by the following lemma:

Lemma 6.2 (Buffer types) *The type of the buffer operations are given by:*

- $r :\uparrow [\mathit{Buffer}\ \tau] \vdash \mathit{Empty}(r)$;
- $x : \tau, a :\uparrow [], b : \mathit{Buffer}\ \tau \vdash \mathit{Insert}(x, b, a)$;
- $b : \mathit{Buffer}\ \tau, r :\uparrow [\tau] \vdash \mathit{Remove}(b, r)$.

Proof. *Follows similarly to lemma 6.1.* ∎

We can now consider the type translation of stream processors, based on the earlier generic definition of how stream processors are translated to the π-calculus. A stream processor is translated to a process expecting to receive two channels representing its input and output streams. Buffers are used to represent these streams, and so the type translation is as shown in Figure 6.11. We note that the type of the values stored in the buffers is not simply the type of values on the streams, because the values are represented using environment entries.

$$[\![\mathbf{SP}\ \tau\ \tau']\!] \quad = \quad \uparrow [\mathit{Buffer}\ \uparrow\uparrow [\![\tau]\!], \mathit{Buffer}\ \uparrow\uparrow [\![\tau']\!]]$$

Figure 6.11: An encoding of stream processor types

6.5.3 Atomic stream processors

The encoding of atomic stream processors is shown in Figure 6.12. The **NullSP** construct is encoded using the $\lceil P \rceil$ notation, where P is just the inactive process. The translation of the term **GetSP** E is similar to the application of a λ-abstraction. However, the argument to apply to the function E is retrieved from the input stream using the *Remove* process. The result of this application is instantiated with the input and output streams. The third form of atomic stream processor **PutSP** $E\ F$, is encoded by using an environment entry representing the output term. A channel corresponding to this term is sent on the output stream using the *Insert* process. After this has taken place then the continuation stream processor F is evaluated and instantiated with the input and output streams.

$$
\begin{array}{lcl}
[\![\textbf{NullSP}]\!]o & = & \lceil \mathbf{0} \rceil \\
[\![\textbf{GetSP } E]\!]o & = & \lceil (\nu r)(\mathit{Remove}(\mathit{in}, r) \mid \\
 & & \quad r?[x].(\nu b)([\![E]\!]b \mid \\
 & & \quad\quad b?[f].(\nu k)(f![x, k] \mid k\langle \mathit{in}, \mathit{out}\rangle))\rceil \\
[\![\textbf{PutSP } E\ F]\!]o & = & \lceil (\nu x)([\![x := E]\!] \mid \\
 & & \quad (\nu a)(\mathit{Insert}(x, \mathit{out}, a) \mid \\
 & & \quad\quad a?[].(\nu d)([\![F]\!]d \mid d\langle \mathit{in}, \mathit{out}\rangle))\rceil
\end{array}
$$

Figure 6.12: An encoding of atomic stream processors

6.5.4 Composite stream processors

Next, we consider the composite stream processors starting with serial composition, which is encoded by creating an intermediate stream between the two stream processors being composed. The expressions corresponding to the two stream processors are evaluated and the π-calculus channels locating their results are used to instantiate the input and output streams. The rightmost stream processor is instantiated with the overall input stream of the composition for its input stream. Its output stream is instantiated with the new intermediate stream, which is also used as the input stream of the leftmost stream processor. Finally, the output stream of the leftmost stream processor is instantiated with the overall output stream of the composition. The entire encoding is shown in Figure 6.13.

$$
\begin{array}{lcl}
[\![E >==< F]\!]o & = & \lceil (\nu l, r, s)([\![E]\!]l \mid \\
 & & \quad [\![F]\!]r \mid \\
 & & \quad \mathit{Empty}(s) \mid \\
 & & \quad s?[\mathit{mid}].(l\langle \mathit{mid}, \mathit{out}\rangle \mid r\langle \mathit{in}, \mathit{mid}\rangle)\rceil
\end{array}
$$

Figure 6.13: An encoding of serial composition

Parallel composition requires the input stream to be split into two duplicate streams. Assuming we can construct a process to accomplish this splitting of streams then the encoding of parallel composition is as shown in Figure 6.14. The two stream processors being composed are evaluated to obtain channels corresponding to them. Next, two new buffers are constructed and used in the to

instantiate the input streams of the two stream processors. These two buffers result from the $Split(in, l, r)$ process which splits the input stream represented by the channel in into two duplicate streams represented by the channels l and r, such that any values sent on the in stream are forwarded to both of the duplicate streams.

$$\begin{aligned}[\![E >\!*\!< F]\!]o \quad = \quad & \lceil(\nu s, t)([\![E]\!]s \mid [\![F]\!]t \mid (\nu lb, rb)(Empty(lb) \mid Empty(rb) \mid \\ & \qquad lb?[l].rb?[r].(Split(in, l, r) \mid \\ & \qquad\qquad s\langle l, out\rangle \mid t\langle r, out\rangle))))\rceil\end{aligned}$$

Figure 6.14: An encoding of parallel composition

Using the definition of unbounded buffers from the previous section we can define the *Split* process. This process reads some input from the specified input stream, in, and continues by inserting the value read into the two buffers specified by l and r. Once both insertions have been acknowledged then the *Split* process recursively calls itself. It must wait for both acknowledgements to ensure that the order of values in the two buffers corresponds to that of the input stream in. The overall definition of the *Split* process is shown in Figure 6.15.

$$\begin{aligned}Split(in, l, r) \quad = \quad & (\nu t)(Remove(in, t) \mid t?[x].(\nu a)(Insert(x, l, a) \mid \\ & \qquad\qquad Insert(x, r, a) \mid \\ & \qquad\qquad a?[].a?[].Split(in, l, r)))\end{aligned}$$

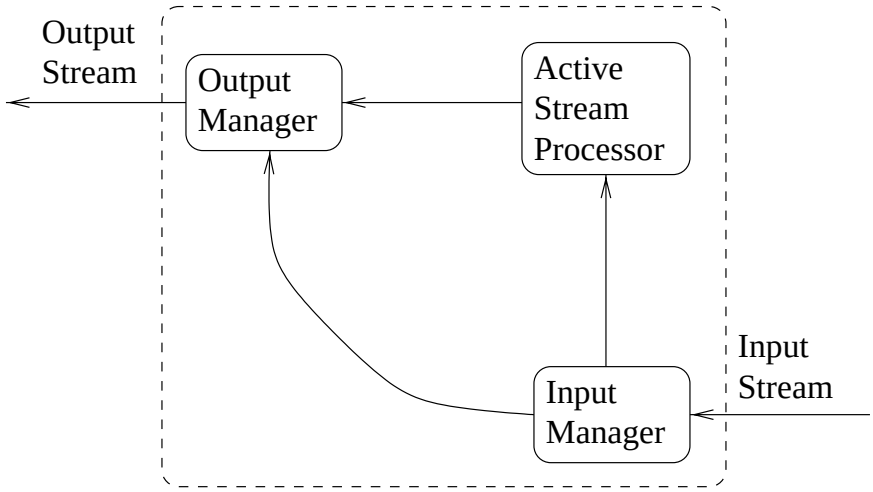
Figure 6.15: Splitting of streams

Next, we consider the looping construct **LoopSP**, which also requires the use of unbounded buffers. The stream processor in the feedback loop is evaluated and instantiated with the overall input stream and an output stream corresponding to a new buffer. This new buffer is split into two separate buffers corresponding to the overall output stream and the overall input stream by using the *Split* process. The entire encoding of feedback is shown in Figure 6.16.

$$[\![\textbf{LoopSP}\ E]\!]o \quad = \quad \lceil(\nu ob)([\![E]\!]s \mid Empty(ob) \mid ob?[b].(s\langle in, b\rangle \mid Split(b, out, in))\rceil$$

Figure 6.16: An encoding of feedback

Finally, we must consider the encoding of the dynamic construct, **DynSP**. The idea used in the encoding is to introduce two extra processes, an input manager and an output manager that handle the input and output streams, and also the communication with the active stream processor:



The input manager analyses values of the sum type on the overall input stream and if they are right injections then they are considered as input for the currently active stream processor. Otherwise the value is a new stream processor, which will be used to replace the currently active stream processor.

The output manager's job is to receive output from the currently active stream processor and forward it to the overall output stream. It also may receive messages from the input manager indicating that the currently active stream processor has changed. Whenever this occurs then the output manager must ignore output from the previously active stream processor and start forwarding the output from the newly active stream processor to the overall output stream.

The encoding of the dynamic construct creates two new buffers which are used to instantiate the initial stream processors input and output streams. In parallel

with this the input and output manager are started and supplied with the overall input and output streams and the two buffers corresponding to the streams connected to the initial stream processor.

$$
\begin{array}{lcl}
[\![\textbf{DynSP}\ E]\!]o & = & \lceil(\nu s)([\![E]\!]s \mid (\nu t,u,n)(Empty(t) \mid Empty(u) \mid \\
 & & \qquad t?[ain].u?[aout].(s\langle ain, aout\rangle \mid \\
 & & \qquad IM(in,ain,n) \mid OM(out,aout,n)))\rceil \\
IM(in,ain,n) & = & Remove(in,s) \mid \\
 & & s?[x].(\nu l,r)(x![l,r] \mid \\
 & & \qquad l?[v].(\nu t,u)(Empty(t) \mid Empty(u) \mid \\
 & & \qquad\quad t?[ain].u?[aout].(v![sp] \mid \\
 & & \qquad\qquad sp\langle ain, aout\rangle \mid \\
 & & \qquad\quad n![aout].IM(in,ain,n))) \mid \\
 & & \qquad r?[y].Insert(ain,y,a) \mid a?[].IM(in,ain,n)) \\
OM(out,aout,n) & = & Remove(aout,r) \mid \\
 & & (r?[x].(Insert(out,x,a) \mid a?[].OM(out,aout,n)) + \\
 & & \ n?[aout].OM(out,aout,n))
\end{array}
$$

Figure 6.17: An encoding of dynamic stream processors

The output manager has to process input from two locations, the output buffer from the active stream processor and notifications from the input manager indicating a change in the active stream processor. The encoding uses the choice construct of the π-calculus to decide at any one time which of these two possible sources of input to read from. This is the only place in the semantics where the choice construct is used, and it is an instance of guarded choice since both branches begin by performing input from a channel variable. This yields a semantics where the currently active stream processor can continue to produce output for a indefinite amount of time before being replaced by a new stream processor. However, the encoding does guarantee that, after an input message requesting a change of the active stream processor, all following input will be processed by the new stream processor.

The following proposition shows that the type structure of the Core Fudgets language is preserved by the encoding into the π-calculus. The type of the channel a is equivalent to $\uparrow [\![\tau]\!]$ up to type equality, as this corresponds to the location of the result of evaluating the term E which we know has type τ.

Proposition 6.1 (Type soundness) *If* $, \vdash E : \tau$ *then* $[\![,]\!], a : \tau' \vdash [\![E]\!]a$ *and* $\tau' \simeq \uparrow [\![\tau]\!]$.

Proof. *The proof proceeds by induction on the structure of the derivation of* $, \vdash E : \tau$. *We show three cases. The other cases are similar, with some using Lemmas 6.1 and 6.2 to derive appropriate types for the list and buffer operations used in the encoding of term* E.

Case $, \vdash \mathbf{n} : \mathbf{num}$ *where* $\mathbf{n} \in \{\mathbf{0}, \mathbf{1}, ...\}$. *From the (Output) rule we have the derivation* $[\![,]\!], a :\uparrow [\![\mathbf{num}]\!] \vdash a![n]$ *because* $[\![\mathbf{num}]\!] = \mathbf{num}$, *and thus we obtain the derivation* $[\![,]\!], a :\uparrow [\![\mathbf{num}]\!] \vdash [\![\mathbf{n}]\!]a$ *as required.*

Case $, \vdash \lambda x.E : \tau \to \tau'$ *where* $, , x : \tau \vdash E : \tau'$. *By induction we have the derivation* $[\![, , x : \tau]\!], b :\uparrow [\![\tau']\!] \vdash [\![E]\!]b$, *and by using weakening we can show that* $[\![,]\!], f :\uparrow [\uparrow\uparrow [\![\tau]\!], \uparrow [\![\tau']\!]], x :\uparrow\uparrow [\![\tau]\!], b :\uparrow [\![\tau']\!] \vdash [\![E]\!]b$. *From the (Input) and (Repl) rules we can show that* $[\![,]\!], f :\uparrow [\uparrow\uparrow [\![\tau]\!], \uparrow [\![\tau']\!]] \vdash *f?[x, b].[\![E]\!]b$. *Using the (Output) rule we have that* $[\![,]\!], a :\uparrow [\![\tau \to \tau']\!], f :\uparrow [\uparrow\uparrow [\![\tau]\!], \uparrow [\![\tau']\!]] \vdash a![f]$ *because* $[\![\tau \to \tau']\!] = \uparrow [\uparrow\uparrow [\![\tau]\!], \uparrow [\![\tau']\!]]$. *Finally, using weakening, the (Prl) and (Res) rules we can show that* $[\![,]\!], a :\uparrow [\![\tau \to \tau']\!] \vdash (\nu f)(a![f] \mid *f?[x, b].[\![E]\!]b)$ *as required.*

Case $, \vdash \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'$. *Using weakening and the (Input) rule we can show that* $[\![,]\!], a :\uparrow [\![\mathbf{SP}\ \tau\ \tau']\!], s : [\![\mathbf{SP}\ \tau\ \tau']\!] \vdash s?[in, out].\mathbf{0}$. *Now, since* $[\![\mathbf{SP}\ \tau\ \tau']\!] =\uparrow [Buffer\ \tau, Buffer\ \tau']$ *then by using the (Output) rule we have that* $[\![,]\!], a :\uparrow\uparrow [Buffer\ \tau, Buffer\ \tau'], s :\uparrow [Buffer\ \tau, Buffer\ \tau'] \vdash a![s]$. *Thus by using the (Prl) and (Res) rules we obtain the derivation* $[\![,]\!], a :\uparrow\uparrow [Buffer\ \tau, Buffer\ \tau'] \vdash (\nu s)(a![s] \mid s?[in, out].\mathbf{0})$ *which is equivalent to* $[\![,]\!], a :\uparrow [\![\mathbf{SP}\ \tau\ \tau']\!] \vdash \lceil \mathbf{0} \rceil$ *as required.*

■

This proof of type soundness is quite lengthy and would benefit from machine checking. This can be accomplished by building a type inference tool for the variant of the *pi*-calculus being used.

Chapter 7

Reasoning about Core Fudgets

In the last chapter we described a formal semantics for Core Fudgets by using an encoding into the π-calculus. This chapter reviews the theory of the π-calculus and describes how we can use it to reason about Core Fudget programs.

Processes in the π-calculus can be compared using a behavioural equivalence. The most often used method for defining such equivalences is the mathematical idea of *bisimulation*. This equates two processes if their transition trees have the same structure and the labels of the transitions are comparable. In Section 7.1 we review this approach to defining equivalences for the π-calculus. The main contribution of this chapter, in Section 7.1.2, is the definition of an equational theory for Core Fudgets. This theory equates two Core Fudget terms if their corresponding π-calculus encodings are bisimilar. This is illustrated by some examples in Section 7.2 where it is used to reason about Core Fudget programs. The chapter concludes with a discussion of the advantages and disadvantages of using the π-calculus to define the semantics of the Core Fudgets language, and some alternative decisions that could be taken in giving Core Fudgets a semantics.

7.1 Bisimulation for the π-calculus

The standard method of defining behavioural equivalences for process calculi is to use bisimulations. A bisimulation on processes is an equivalence based on the observable properties of the processes. However, there are many different ways of defining the observable properties which lead to numerous formulations of bisimulation.

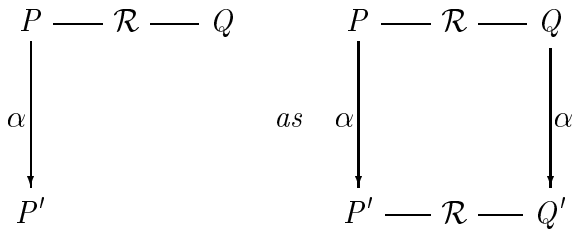
The original definition of bisimulation is due to Park [Par81] and was developed as a method for comparing automata. However, Milner later realized that this idea of bisimulation could be used to formulate a precise behavioural equivalence for the process calculus CCS [Mil85]. This is where we find the classical form of bisimulation, which in the context of the π-calculus is often referred to as *ground bisimulation*. This compares processes based purely upon the actions that they can perform. However, because the π-calculus we presented in Chapter 5 has an associated typing system then we must introduce the concept of a *confined relation* before we define ground bisimulation formally:

Definition 7.1 (Confined relation) *A set of relations, $\mathcal{R}_\Gamma$ between processes with the same context , is a* confined relation *if $P \mathcal{R}_\Gamma Q$ implies , $\vdash P$ and , $\vdash Q$. We shall write , $\vdash P \mathcal{R} Q$ to mean that $P \mathcal{R}_\Gamma Q$, and usually omit any obvious type information.*

Definition 7.2 (Ground simulation) *A confined relation $\mathcal{R}$ between processes is a* ground simulation *if , $\vdash P \mathcal{R} Q$ implies:*

- *if , $\vdash P \xrightarrow{\alpha} P'$ then $\exists Q'.$, $\vdash Q \xrightarrow{\alpha} Q'$ and , $\vdash P' \mathcal{R} Q'$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form (for simplicity we omit typing information):

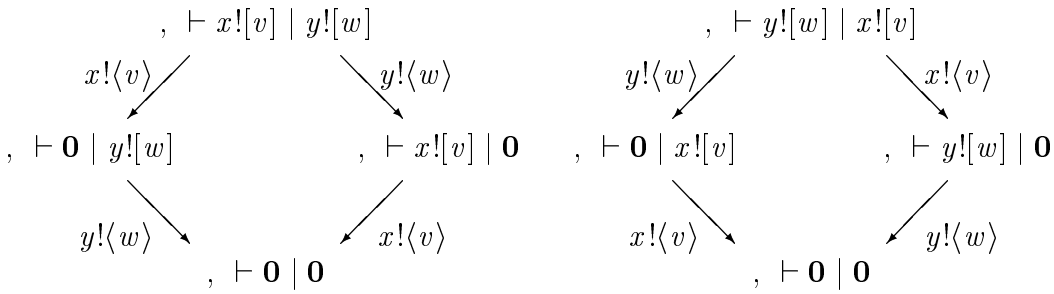


Definition 7.3 (Ground bisimulation) *A confined relation, $\mathcal{S}$, is a* ground bisimulation *if $\mathcal{S}$ and $\mathcal{S}^{-1}$ are ground simulations. Ground bisimilarity, $\sim$, is defined to be the greatest such relation.*

Here α corresponds to an action, and $P \xrightarrow{\alpha} P'$ means that the process P can evolve by performing the transition labelled by the action α to the process P'. The syntax of actions is determined by the process calculi that we are using. For example, in

CCS actions only express synchronisations, but in the π-calculus actions can capture the communication of channel names and so syntactic equivalence of actions must be defined up to α-conversion.

We now illustrate ground bisimulation with a simple example from the π-calculus. The transition trees for the two processes $, \vdash x![v].\mathbf{0} \mid y![w].\mathbf{0}$ and $, \vdash y![w].\mathbf{0} \mid x![v].\mathbf{0}$ are as follows:



It is straightforward to see that these trees are equivalent modulo ordering of branches. This property is captured by ground bisimulation and so we can conclude that these two processes are ground bisimilar.

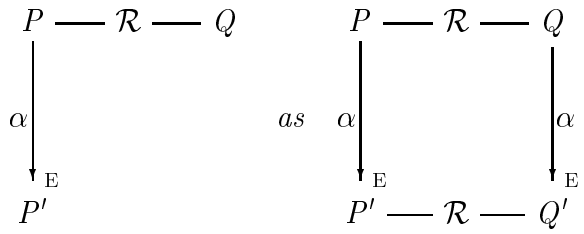
Two different approaches have been developed to adopt this technique for value-passing calculi, called *early* and *late*, respectively. Intuitively, the difference between these two approaches is how input actions are dealt with in the definition of bisimulation. In the early approach, the variable bound by an input action is instantiated at the moment of inferring an input action, as in the semantics presented in Section 5.4. However, in the late approach, the instantiation of the bound variable occurs when a communication is inferred.

The distinction between early and late can also be seen in the transition semantics for the calculus. In the π-calculus an early transition system has inference rules for input and communication of the form shown in Figure 7.1. The transition relation, $\xrightarrow{\alpha}_E$, is subscripted with E to indicate that it is an early transition relation. In this case, the transition semantics quantifies over the possible instantiations for bound variables in input prefixes. A process that can perform an input action will have an infinite family of transitions corresponding to the different instantiations of the bound variable. In the late approach this instantiation is not made in the transition system but must be captured explicitly in the definition of bisimulation. Using the idea of bisimulation from CCS we can define an equivalence using the early semantics as:

Definition 7.4 (Early simulation) *A confined relation $\mathcal{R}$ between processes is an early simulation if $, \vdash P\mathcal{R}Q$ implies:*

- *if $, \vdash P \xrightarrow{\alpha}_E P'$ then $\exists Q'., \vdash Q \xrightarrow{\alpha}_E Q'$ and $, \vdash P'\mathcal{R}Q'$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form (for simplicity we omit typing information):



Definition 7.5 (Early bisimulation) *A confined relation, $\mathcal{S}$, is an early bisimulation if $\mathcal{S}$ and $\mathcal{S}^{-1}$ are early simulations. Early bisimulation, $\sim_E$, is defined to be the greatest such relation.*

$$(In) \quad \frac{, \vdash \vec{v} : \vec{\tau} \quad , (x) \simeq \uparrow [\vec{\tau}]}{, \vdash x?[\vec{y}].P \xrightarrow{x?\langle \vec{z} \rangle}_E P[\vec{z}/\vec{y}]}$$

$$(Com) \quad \frac{, \vdash P \xrightarrow{(\nu \vec{y'})x!\langle \vec{y} \rangle}_E P' \quad , \vdash Q \xrightarrow{x?\langle \vec{y} \rangle}_E Q'}{, \vdash P \mid Q \xrightarrow{\tau}_E (\nu \vec{y'})(P' \mid Q')} \quad \vec{y'} \cap fn(Q) = \emptyset$$

Figure 7.1: An early transition semantics

The corresponding late semantics is characterised by the inference rules shown in Figure 7.2. The definition of late bisimulation is complicated by the fact that input actions need to be treated as a special case:

Definition 7.6 (Late simulation) *A confined relation $\mathcal{R}$ between processes is a late simulation if $, \vdash P\mathcal{R}Q$ implies:*

- *if $, \vdash P \xrightarrow{a?\langle \vec{y} \rangle}_L P'$ then $\exists Q'., \vdash Q \xrightarrow{a?\langle \vec{y} \rangle}_L Q'$ and $\forall \vec{z}., \vdash P'[\vec{z}/\vec{y}]\mathcal{R}Q'[\vec{z}/\vec{y}]$;*
- *if $, \vdash P \xrightarrow{\alpha}_L P'$ then $\exists Q'., \vdash Q \xrightarrow{\alpha}_L Q'$ and $, \vdash P'\mathcal{R}Q'$ for $\alpha \neq a?\langle \vec{y} \rangle$.*

Definition 7.7 (Late bisimulation) *A confined relation, $\mathcal{S}$, is a late bisimulation if $\mathcal{S}$ and $\mathcal{S}^{-1}$ are late simulations. Late bisimulation, $\sim_L$, is defined to be the greatest such relation.*

$$(In) \qquad , \vdash x?[\vec{y}].P \xrightarrow{x?\langle\vec{y}\rangle}_L P$$

$$(Com) \qquad \frac{, \vdash P \xrightarrow{(\nu\vec{y'})x!\langle\vec{z}\rangle}_L P' \quad , \vdash Q \xrightarrow{x?\langle\vec{y}\rangle}_L Q'}{, \vdash P \mid Q \xrightarrow{\tau}_L (\nu\vec{y'})(P' \mid Q'[\vec{z}/\vec{y}])} \quad \vec{y'} \cap fn(Q) = \emptyset$$

Figure 7.2: A late transition semantics

In this dissertation, we will make use of early bisimulation. Our motivation for this choice is twofold. Firstly, the transition system for the semantics of the π-calculus in Section 5.4 facilitates a simpler definition of early bisimulation than late bisimulation. In the late case, input actions must be dealt with separately in the definition of bisimulation whereas in the early case a single clause is sufficient. Secondly, the intuition behind early bisimulation is easier to understand, as the instantiation of a bound variable occurs simultaneously with the choice of where the input is occuring, whereas late bisimulation separates these two events.

To use bisimulation as the basis for an equational theory, where one expression can be replaced by another that is semantically equivalent, it must be preserved by as many of the operators of the language as possible. We can show that early bisimulation is preserved by all of the following operators:

Proposition 7.1 *If $, \vdash P \sim_E Q$ then we have the following:*

1. *if $, (x) \simeq \vec{\tau}$ and $, (\vec{v}) = \vec{\tau}$ then $, \vdash x![\vec{v}].P \sim_E x![\vec{v}].Q$;*
2. *$, \vdash (\nu\vec{x})P \sim_E (\nu\vec{x})Q$;*
3. *if P and Q are guarded processes then if $, \vdash G$ then $, \vdash G \mid P \sim_E G \mid Q$ and $, \vdash P \mid G \sim_E Q \mid G$;*
4. *if $, \vdash R$ then $, \vdash R + P \sim_E R + Q$ and $, \vdash P + G \sim_E Q + G$;*
5. *$, \vdash *P \sim_E *Q$;*

6. *if* , $(v) \simeq \mathbf{num}$ *and* , $\vdash R$ *then* , $\vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ R \sim_E \mathbf{if}\ v\ \mathbf{then}\ Q\ \mathbf{else}\ R$ *and* , $\vdash \mathbf{if}\ v\ \mathbf{then}\ R\ \mathbf{else}\ P \sim_E \mathbf{if}\ v\ \mathbf{then}\ R\ \mathbf{else}\ Q$*;*

7. *if* , $(v) \simeq \mathbf{num}$ *and* , $(w) \simeq \mathbf{num}$ *and also*

8. *if* , $(v) \simeq \mathbf{num}$ *and* , $(w) \simeq \mathbf{num}$ *and also* , $(x) \simeq\uparrow [\mathbf{bool}]$ *then it follows that* , $\vdash \mathbf{equal}\ [v, w, x].P \sim_E \mathbf{equal}\ [v, w, x].Q$.

9. *if* , $(v) \simeq \mathbf{num}$ *and* , $(w) \simeq \mathbf{num}$ *and also* , $(x) \simeq\uparrow [\mathbf{num}]$ *then it follows that* , $\vdash \mathbf{add}\ [v, w, x].P \sim_E \mathbf{add}\ [v, w, x].Q$.

Proof. *The proof is standard and follows the corresponding proof for the original π-calculus [MPW89]. We only consider output prefix. Define:*

$$\mathcal{S}_\Gamma = \{(x![\vec{v}].P, x![\vec{v}].Q) \mid , \vdash P \sim_E Q\} \cup \sim_E .$$

We proceed by a case analysis on the rule used to infer a transition for , $\vdash x![\vec{v}].P$. *There is only one possibility due to the* (Out) *transition rule, and we have:*

$$, \vdash x![\vec{v}].P \xrightarrow{x!\langle\vec{v}\rangle} P.$$

From the (Out) *rule we also have that* , $\vdash x![\vec{v}].Q \xrightarrow{x!\langle\vec{v}\rangle} Q$, *and because* , $\vdash P \sim_E Q$ *then* $P \mathcal{S}_\Gamma Q$ *as required. The other parts follow in a similar manner, with the technique of bisimulation up to restriction, as used by Milner, Parrow and Walker [MPW89], useful in proving part (3).* ∎

Although early bisimulation is preserved by most of the operators of the language, it is not preserved by input prefixing. This is illustrated by the following two π-calculus processes:

$$\begin{aligned} P &= x?[] \mid y![] \\ Q &= x?[].y![] + y![].x?[] \end{aligned}$$

which are obviously related by early bisimilarity. Unfortunately though, if we prefix the above two processes by an input yielding the processes $a?[x].P$ and $a?[x].Q$ then these processes are not early bisimilar. This is because there is a transition for the first process, $a?[x].P \xrightarrow{a?\langle y\rangle} y?[] \mid y![] \xrightarrow{\tau} \mathbf{0}$, which cannot be matched by the second, $a?[x].Q \xrightarrow{a?\langle y\rangle} y?[].y![] + y![].y?[] \not\xrightarrow{\tau}$. Milner, Parrow and Walker solve this problem by quantifying over all possible substitutions for free channel names, resulting in early congruence:

Definition 7.8 (Early congruence) *If* $\Gamma, \Delta = \vec{x} : \vec{\tau}$, *then we define a* Δ*-substitution to be a substitution* $\cdot[\vec{v}/\vec{x}]$ *where* $\Gamma \vdash \vec{v} : \vec{\tau}$. *Early congruence,* $\sim_E^c$, *is the least confined relation such that:*

- $\Gamma, \Delta \vdash P \sim_E^c Q$ *iff* $\Gamma, \Delta \vdash P[\vec{v}/\vec{x}] \sim_E Q[\vec{v}/\vec{x}]$ *for all* Δ*-substitutions* $\cdot[\vec{v}/\vec{x}]$.

Relations will be denoted to be congruences by the addition of a superscript 'c'.

Proposition 7.2 *If* $\Gamma, \Delta \vdash P \sim_E^c Q$ *then* $\Gamma, \Delta \vdash x?[\vec{y}].P \sim_E^c x?[\vec{y}].Q$.

Proof. *We show that the following relation is an early congruence:*

$$\mathcal{S}_\Gamma = \{(x?[\vec{y}].P, x?[\vec{y}].Q) \mid \Gamma, \Delta \vdash P \sim_E^c Q\}.$$

Proceeding by case analysis on the transition rule used to infer a transition for $\Gamma, \Delta \vdash x?[\vec{y}].P$, *we have only one possibility, due to the* (In) *rule:*

$$\frac{\Gamma, \Delta \vdash v_i : \tau_i \text{ for } i \in \{1..n\} \quad \Gamma, \Delta(x) \simeq\uparrow [\vec{\tau}]}{\Gamma, \Delta \vdash x?[\vec{y}].P \xrightarrow{x?\langle \vec{v} \rangle} P[\vec{v}/\vec{y}]} .$$

Similarly, we can derive the transition $\Gamma, \Delta \vdash x?[\vec{y}].Q \xrightarrow{x?\langle \vec{v} \rangle} Q[\vec{v}/\vec{y}]$. *However, because* $\Gamma, \Delta \vdash P \sim_E^c Q$ *then it must be that* $\Gamma, \Delta \vdash P[\vec{v}/\vec{y}] \mathcal{S} Q[\vec{v}/\vec{y}]$ *as required.* ∎

It follows from the definition of $\sim_E^c$ that any two terms related by $\sim_E$ are also related by $\sim_E^c$ and thus proposition 7.1 holds for $\sim_E^c$. Using this with proposition 7.2 we can conclude that early congruence is preserved by all of the operators of the π-calculus. This is important as it allows us to consider using early congruence as the basis of an equational theory, where we can replace a subterm of a process with another early congruent subterm.

An alternative approach to obtaining a relation that is preserved by all operators of the language is the concept of *open bisimulation* studied by Sangiorgi [San93]. This integrates the instantiation of names into the definition of bisimulation in such a manner that it is preserved by all operators of the language. We do not consider open bisimulation further in this dissertation, although it could be used to prove all of the results we consider in Section 7.2. Whether it simplifies these proofs is an open question, as the size of relations required to define open bisimulations can be large, although its complexity may be lower than that of early congruence.

7.1.1 Weak bisimulations

The bisimulations reviewed so far have all been *strong bisimulations*. They all consider the internal moves of processes – the communication actions – in the same manner as the other actions corresponding to the external behaviour of the processes. This results in rather strict relations, and it is often the case that we wish to abstract away from the internal details of processes.

Bisimulations that abstract away from communication actions which represent the internal behaviour of processes are called *weak bisimulations*. They correspond to a weak behavioural equivalence that equates processes only based upon their external behaviour and not the internal behaviour of the processes. The standard formulation of these bisimulations involves defining a concept of *weak transitions*:

Definition 7.9 (Weak transitions) *Let* $\Longrightarrow$ *be the reflexive and transitive closure of* $\stackrel{\tau}{\longrightarrow}$*, and* $\stackrel{\alpha}{\Longrightarrow}$ *be the relational composition* $\Longrightarrow \circ \stackrel{\alpha}{\longrightarrow} \circ \Longrightarrow$*. A weak transition,* $\stackrel{\widehat{\alpha}}{\Longrightarrow}$ *is defined as:*

$$\stackrel{\widehat{\alpha}}{\Longrightarrow} = \begin{cases} \stackrel{\alpha}{\Longrightarrow} & \text{if } \alpha \neq \tau \\ \Longrightarrow & \text{otherwise} \end{cases}$$

A weak bisimulation is defined by requiring that strong transitions are matched by weak ones. For example, the definition of weak early bisimilarity is given by:

Definition 7.10 (Weak early simulation) *A confined relation* $\mathcal{R}$ *between processes is a weak early simulation if* , $\vdash P \mathcal{R} Q$ *implies:*

- *whenever* , $\vdash P \stackrel{\alpha}{\longrightarrow}_E P'$ *then* $\exists Q'.$, $\vdash Q \stackrel{\alpha}{\Longrightarrow}_E Q'$ *and* , $\vdash P' \mathcal{R} Q'$.

Definition 7.11 (Weak early bisimilarity) *A confined relation,* $\mathcal{S}$*, is a weak early bisimulation if* $\mathcal{S}$ *and* $\mathcal{S}^{-1}$ *are weak early simulations. Weak early bisimulation* $\approx_E$ *is defined to be the greatest such relation.*

In a similar manner to strong early bisimilarity, we must explicitly define a congruence to handle the fact that weak early bisimilarity is not preserved by input prefixes. We denote the largest weak early congruence as $\approx_E^c$, which is defined analogously to the early congruence of Definition 7.8.

Proposition 7.3 *Weak early congruence $\approx_E^c$ is preserved by all of the operators of the π-calculus.*

Proof. *Follows in a similar manner to propositions 7.1 and 7.2.* ∎

If we allowed unrestricted choice in our variant of the π-calculus then this proof fails. However, the encoding of Core Fudgets in Chapter 6 only makes use of guarded choice and so we only included guarded choice in our π-calculus variant to ensure that this proof does not fail.

7.1.2 Equivalence for Core Fudgets

The theory presented in this chapter so far is standard for the π-calculus. In this section we make the main contribution of this chapter by defining an equational theory for Core Fudgets. We propose using weak early congruence as an equivalence for the Core Fudgets language. A weak equivalence is preferable over a strong one as we do not wish to distinguish between Core Fudget terms that only differ in their internal behaviour. For example, intuitively we would expect the following terms to be equivalent:

$$\textbf{PutSP}\ x\ \textbf{NullSP}$$
$$\textbf{GetSP}\ (\lambda y.\textbf{PutSP}\ x\ \textbf{NullSP}) >==<\ \textbf{PutSP}\ z\ \textbf{NullSP}.$$

However, a strong bisimulation does not relate these terms and hence is not an appropriate choice for the basis of an equivalence on Core Fudget terms. A weak bisimulation will relate these terms, capturing the fact that we do not wish the observer to discriminate terms based on their internal behaviour. We can now define an equivalence on Core Fudget terms by introducing an equational theory:

Definition 7.12 (Equational theory) *An equational theory of Core Fudgets, $\mathcal{T}_{cf}$, consists of equations of the form , $\vdash E = F : \tau$, asserting that the two terms E and F are equivalent. Formally, we define this equivalence relation as:*

- , $\vdash E = F : \tau$ *if and only if* $[\![,]\!], a :\uparrow [\![\tau]\!] \vdash [\![E]\!]a \approx_E^c [\![F]\!]a$.

It is straightforward to check that weak early bisimulation, and hence weak early congruence, is actually an equivalence.

Proposition 7.4 *Weak early bisimulation, $\approx_E$, is an equivalence relation.*

Proof. *By definition $\approx_E$ is symmetric, and it is straightforward to show $\{(E, E) \mid , \vdash E \approx_E E\}$ to be a weak early bisimulation thus proving reflexivity. Similarly, we prove transitivity by showing $\approx_E \circ \approx_E$ to be a weak early bisimulation.* ∎

7.2 Examples

In this section we illustrate the equational theory from Definition 7.12 by formally analysing some Core Fudget programs. We start by revisiting the identity stream processor example from Section 4.2.1, and examine its semantics according to the π-calculus encoding of Core Fudgets.

7.2.1 The identity stream processor

We recall the definition of the identity stream processor in Core Fudgets:

$$idSP = \textbf{Fix}\ i.(\textbf{GetSP}\ (\lambda x.\textbf{PutSP}\ x\ i)).$$

According to the π-calculus definition, the semantics of this program is given by the following process:

$$\begin{aligned}
&[\![idSP]\!]o \\
&= (\nu i)([\![i := \textbf{GetSP}\ (\lambda x.\textbf{PutSP}\ x\ i)]\!] \mid \\
&\qquad [\![\textbf{GetSP}\ (\lambda x.\textbf{PutSP}\ x\ i)]\!]o) \\
&= (\nu i)(*i?[c].[\![\textbf{GetSP}\ (\lambda x.\textbf{PutSP}\ x\ i)]\!]c \mid \\
&\qquad [\![\textbf{GetSP}\ (\lambda x.\textbf{PutSP}\ x\ i)]\!]o).
\end{aligned}$$

We now concentrate on the body of the definition because it occurs in both of the branches of the parallel composition. The encoding of this term yields the process:

$$\begin{aligned}
&[\![\textbf{GetSP}\ (\lambda x.\textbf{PutSP}\ x\ i)]\!]o \\
&= \lceil(\nu r)(Remove(in, r) \mid \\
&\qquad r?[x].(\nu b)([\![\lambda x.\textbf{PutSP}\ x\ i]\!]b \mid \\
&\qquad\qquad b?[f].(\nu k)(f![x, k] \mid k\langle in, out\rangle))))\rceil \\
&= \lceil(\nu r)(Remove(in, r) \mid \\
&\qquad r?[x].(\nu b)((\nu f)(b![f] \mid *f?[x, b][\![\textbf{PutSP}\ x\ i]\!]b) \mid \\
&\qquad\qquad b?[f].(\nu k)(f![x, k] \mid k\langle in, out\rangle))))\rceil.
\end{aligned}$$

The semantics of the π-calculus does not allow us to reduce the communication on the channel b because it is guarded by an input prefix. However, using the weak early congruence described in the previous section we can reduce the communication but weakening the statement to indicate the equivalence as weak early congruence:

$$\begin{aligned}\approx^c_E \quad & \lceil(\nu r)(Remove(in, r) \mid \\ & \quad r?[x].(\nu f)(*f?[x, b][\![\mathbf{PutSP}\ x\ i]\!]b \mid \\ & \quad\quad (\nu k)(f![x, k] \mid k\langle in, out\rangle)))\rceil.\end{aligned}$$

Similarly, we can now consider the reduction of the communication possible on the channel f. The replicated input process on f will never be able to receive any input after this communication and can also be eliminated, yielding the equivalent process:

$$\begin{aligned}\approx^c_E \quad & \lceil(\nu r)(Remove(in, r) \mid \\ & \quad r?[x].(\nu k)([\![\mathbf{PutSP}\ x\ i]\!]k \mid k\langle in, out\rangle))\rceil.\end{aligned}$$

Now, we can expand out the remaining Core Fudget term and the stream processor instantiation $k\langle in, out\rangle$, to obtain the process:

$$\begin{aligned}= \quad & \lceil(\nu r)(Remove(in, r) \mid \\ & \quad r?[x].(\nu k)((\nu s)(k![s] \mid \\ & \quad\quad s?[in, out].(\nu y)([\![y := x]\!] \mid \\ & \quad\quad\quad (\nu a)(Insert(y, out, a) \mid \\ & \quad\quad\quad\quad a?[].(\nu d)([\![i]\!]d \mid d\langle in, out\rangle))))) \mid \\ & \quad\quad k?[s].s![in, out]))\rceil.\end{aligned}$$

Note that we have not used the simpler notation $\lceil\_\rceil$ in the expansion of the *bfPutSP* term as this would have hidden the communication that is possible on the channel k. Again, we consider the reduction corresponding to this communication that yields an equivalent process up to weak early congruence:

$$\begin{aligned}\approx^c_E \quad & \lceil(\nu r)(Remove(in, r) \mid \\ & \quad r?[x].(\nu s)(s?[in, out].(\nu y)([\![y := x]\!] \mid \\ & \quad\quad (\nu a)(Insert(y, out, a) \mid \\ & \quad\quad\quad a?[].(\nu d)([\![i]\!]d \mid d\langle in, out\rangle))) \mid \\ & \quad\quad s![in, out]))\rceil.\end{aligned}$$

Once more, we have a possible reduction corresponding to the communication on the channel s, and this results in the equivalent process:

$$\begin{array}{ll} \approx_E^c & \lceil(\nu r)(Remove(in, r) \mid \\ & \quad r?[x].(\nu y)([\![y := x]\!] \mid \\ & \quad\quad (\nu a)(Insert(y, out, a) \mid \\ & \quad\quad\quad a?[].(\nu d)([\![i]\!]d \mid d\langle in, out\rangle)))))\rceil. \end{array}$$

We can eliminate the use of the environment entry as the channel x corresponds to a term removed from the input stream. All the channels communicated along input streams correspond to triggers for evaluating the actual terms being sent, and so the environment entry in the above process just introduces an extra level of indirection:

$$\begin{array}{ll} \approx_E^c & \lceil(\nu r)(Remove(in, r) \mid \\ & \quad r?[x].(\nu a)(Insert(x, out, a) \mid \\ & \quad\quad a?[].(\nu d)([\![i]\!]d \mid d\langle in, out\rangle)))))\rceil. \end{array}$$

This simplified process can be replaced into the original expansion of *idSP* where it occurs twice due to the recursive nature of the stream processor. However, the above form of the process clearly illustrates the behaviour of the identity stream processor. Initially it removes an element from the input stream, and then inserts this into the output stream. Once the insertion has been performed it evaluates the term i, corresponding to a recursive call to the stream processor. This call yields a stream processor which is then instantiated with the same input and output streams. The two channels r and a are used to ensure the sequentiality of the reading and writing from the input and to the output stream, respectively.

7.2.2 Commutativity of parallel composition

The next example we inspect corresponds to parallel composition of stream processors being commutative. This is an equational rule that one would expect to hold given the informal description of the semantics stated in Chapter 3. Here, we formally prove this equational rule which can be stated as:

$$, \vdash E >\!*\!< F = F >\!*\!< E : \mathbf{SP}\ \tau\ \tau'.$$

Using the definition of our equational theory this rule is valid if and only if the corresponding encoding into the π-calculus is a correct statement:

$$, , a :\uparrow [\![\mathbf{SP}\ \tau\ \tau']\!] \vdash [\![E >\!*\!< F]\!]a \approx^c_E [\![F >\!*\!< E]\!]a.$$

Starting with the left hand side of this equation we show how we can derive the right hand side:

$$\begin{array}{ll} & [\![E >\!*\!< F]\!]o \\ = & \lceil(\nu s,t)([\![E]\!]s \mid [\![F]\!]t \mid (\nu lb,rb)(Empty(lb) \mid Empty(rb) \mid \\ & \qquad lb?[l].rb?[r].(Split(in,l,r) \mid s\langle l,out\rangle \mid t\langle r,out\rangle)))\rceil. \end{array}$$

By renaming the local channel names, we can show this to be α-equivalent to the following process:

$$\begin{array}{ll} \equiv_\alpha & \lceil(\nu s,t)([\![E]\!]t \mid [\![F]\!]s \mid (\nu lb,rb)(Empty(lb) \mid Empty(rb) \mid \\ & \qquad lb?[l].rb?[r].(Split(in,l,r) \mid t\langle l,out\rangle \mid s\langle r,out\rangle)))\rceil. \end{array}$$

Now, using weak early congruence we can reorder the parallel processes to give:

$$\begin{array}{ll} \approx^c_E & \lceil(\nu s,t)([\![F]\!]s \mid [\![E]\!]t \mid (\nu lb,rb)(Empty(lb) \mid Empty(rb) \mid \\ & \qquad lb?[l].rb?[r].(Split(in,l,r) \mid t\langle l,out\rangle \mid s\langle r,out\rangle)))\rceil. \end{array}$$

which is almost the right hand side we desire, except that the channels l and r are swapped in the processes $s\langle r,out\rangle$ and $t\langle l,out\rangle$. The proof of commutativity can be completed if we can show that:

$$Split(in,l,r) \mid t\langle l,out\rangle \mid s\langle r,out\rangle \approx^c_E Split(in,l,r) \mid s\langle l,out\rangle \mid t\langle r,out\rangle.$$

We proceed by starting with the left hand side of this equation and renaming channel variables, then we can rearrange the parallel processes using weak early congruence:

$$\begin{array}{ll} & Split(in,l,r) \mid t\langle l,out\rangle \mid s\langle r,out\rangle \\ = & \quad Split(in,r,l) \mid t\langle r,out\rangle \mid s\langle l,out\rangle \\ \approx^c_E & \quad Split(in,r,l) \mid s\langle l,out\rangle \mid t\langle r,out\rangle \end{array}$$

The remaining stage of the proof is to show that $Split(in,r,l) \approx^c_E Split(in,l,r)$, and this is straightforward since we can rearrange the *Insert* parallel processes in

the definition of *Split* using weak early congruence. This completes the proof of the original equation we set out to prove, verifying that parallel composition of stream processors is indeed commutative according to the π-calculus semantics.

7.3 Analysis of π-calculus encoding

In this section we consider the advantages and disadvantages of using the π-calculus for specifying the semantics of Core Fudgets. Reasons for using the π-calculus as the basis of a semantics for Core Fudgets were discussed at the start of Chapter 5. Here, we concentrate on the advantages of this approach that became apparent during and after the development of the semantics. Particular emphasis is placed on assessing the usefulness of the theory of the π-calculus for reasoning about Core Fudget terms through the encoding into the π-calculus:

- There is no need to develop a new theory, as the theory of the π-calculus can be drawn upon to induce a theory for Core Fudgets. There are numerous definitions of equivalence for the π-calculus and an appropriate one for Core Fudgets is weak early congruence as described in Section 7.1.1.

- The π-calculus encoding can be used as a reference implementation, by making use of an implementation of the π-calculus. The PICT programming language [PT97], based on the π-calculus, makes a particularly good base for executing the π-calculus semantics. Although it does not provide direct support for the synchronous calculus, the encoding given in Chapter 6 can be modified to use only asynchronous communication. PICT has a type system very similar to that described in Chapter 6, and also supports polymorphism. This is useful as, in practice, most interesting stream processors are polymorphic.

- As no new theory was required, the π-calculus provided a useful means for exploring the possible semantic issues of the Core Fudgets language. This language has some constructs whose precise semantic meaning is not entirely clear, and various possibilities for their meaning exist. The π-calculus provided a formal mechanism for precisely specifying these different alternatives and gave insight into them.

- It is possible to automate the checking of equivalences for the π-calculus using tools such as the Mobility Workbench [VM94]. However, these tools do not tend to work well with recursive π-calculus expressions. This is a problem because the majority of interesting Core Fudget programs involve recursion, whose encodings are recursive π-calculus processes.

Although the π-calculus has proved to be advantageous in developing a semantics for Core Fudgets, there are also some serious disadvantages with this approach:

- The π-calculus is a very low level language in which the only means of computation is communication. This leads to even relatively simple Core Fudget terms having encodings that are large and complicated. The intuition behind the semantics of the various constructs in the Core Fudgets language tend to get lost in the exact details of these encodings.
- The most interesting Core Fudget terms tend to be recursive, and to prove properties about such terms by using the theory of the π-calculus can be complicated. The standard technique for proving bisimulations involving such terms is coinduction [MT90, Pit94], but this can be difficult to apply.

7.4 Semantic issues

During the development of the π-calculus semantics for Core Fudgets it became clear that a number of semantic issues need to be addressed for any semantics of Core Fudgets. Here we consider these issues and their possible solutions.

The original implementation of the Fudgets system evaluates stream processors in a fixed order. This order is chosen to correspond to the demand of data, corresponding to the demand driven nature of the Haskell language due to its laziness. Two stream processors in series, $A >==< B$, are evaluated by first evaluating A. If A can produce some output then this action is undertaken, and the residual of A is recursively evaluated. Only when A requires input to continue is the stream processor B evaluated, and then only far enough to satisfy the demand for data from A. Specifying such a fixed order of evaluation in the semantics avoids the need to consider the wires between stream processors as unbounded buffers.

An alternative semantic choice is to consider no particular evaluation order, which is the decision taken by the semantics in the previous chapter. This has

the effect of requiring the wires between stream processors to be considered as unbounded buffers.

Another place in the semantics where various choices exist is the semantics of the merging operations required in parallel composition and feedback. It is hard to justify any other interpretation of the merge for parallel composition than a nondeterministic merge. However, in the case of feedback there are practical arguments for the merging operation being deterministic. Consider a graphical program consisting of a group of buttons, one of which is always depressed whilst the others are not. When the user presses one of the buttons it becomes the depressed one, and the currently depressed button is raised. One way of modelling this using stream processors is to represent each of the buttons with a single stream processor. All of these stream processors are combined in parallel, and this overall stream processor is then combined using feedback. The feedback is required so that when one of the buttons is pressed it can indicate to all the others that it has just been depressed. The button that was depressed receives the information about the new depression and so can raise itself. However, if we let the overall input to the feedback be used to programmatically depress a button, then we run into problems. This model of the program will only work in a reasonable manner if the merge operation on the input side of feedback biases towards the feedback loop. If not, then it would be possible to receive a programmatic message indicating a depression of a button before the message indicating an actual action by the user depressing a button.

Chapter 8

Operational semantics of Core Fudgets

In the previous chapter we presented a formal semantics for the Core Fudget language using an encoding into the π-calculus. Although this captured the precise meaning of Core Fudget programs the encoding of even trivial Core Fudget programs can be large and unwieldy π-calculus processes. More importantly the intuition behind the semantics is hidden by the detail in these processes. This chapter describes a direct operational semantics for Core Fudgets solving these problems. This semantics is inspired by the lessons learnt from the denotational semantics of Chapter 6, and is based on a labelled transition system. We examine some bisimulation equivalences and finally choose a variant of weak higher-order bisimulation, which we show to be a congruence. The semantics is closely related to the semantics that Ferriera et. al [FHJ95] developed for Concurrent ML. There are some differences between the operational and denotational semantics but we believe these can be easily eliminated yielding the relationship shown in Figure 8.1.

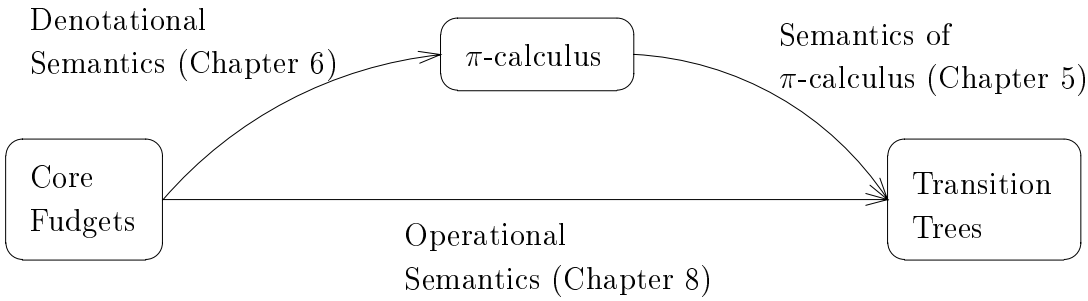


Figure 8.1: The relationship between the semantics

8.1 An Operational semantics

In this section we define a structured operational semantics for Core Fudgets using ideas from Milner's concurrent language CCS [Mil85]. This semantics was inspired by some of the lessons learnt from the denotational semantics described in Chapter 6. We believe that an operational semantics is more intuitive and simpler than the denotational semantics. Insight into different implementations of the Fudgets system is easier with this semantics, and coinductive proofs [MT90, Pit94] are also simpler.

The semantics is defined in terms of a labelled transition system that characterises the immediate observations of Core Fudget programs. A labelled transition system is a triple, $(S, A, \xrightarrow{\alpha})$, comprising a set S of states, a set A of actions used to label transitions, and a transition relation, $\xrightarrow{\alpha} \subseteq S \times S$ for each $\alpha \in A$. We extend this definition to include a typing context, in the same way as was done for the operational semantics of the π-calculus in Section 5.4. The transition relation is defined as , $\vdash E : \tau \xrightarrow{\alpha} F$ meaning that , $\vdash E : \tau$ and E can evolve into F by performing the action α. The typing context is required to ensure that only values of the correct type are communicated along streams.

8.1.1 Feed

Before introducing the operational semantics, we introduce a new construct into the Core Fudgets language, called *feed*. This construct is useful in specifying the operational semantics. We extend the syntax of atomic stream processors to include this new construct as follows:

$$\begin{array}{lcll} S_A, T_A, ... \in SP_A & ::= & ... & \textit{Atomic Stream Processors} \\ & | & E \ll F & \textit{Feed} \end{array}$$

The intuitive meaning of the term $E \ll F$ is to add the term F to the input stream of the stream processor E. In this sense it can be thought of as a dual to **PutSP** E F which corresponds to the stream processor F with E added to its output stream. This relationship can be seen in Figure 8.2. This construct corresponds directly to the *feedSP* construct in the Chalmers implementation of the Fudgets system. The typing rule for this new construct is shown in Figure 8.3.

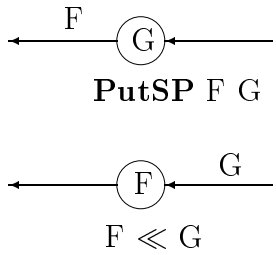


Figure 8.2: The duality of **PutSP** and $\ll$

$$(\textit{Feed}) \quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \quad , \vdash F : \tau}{, \vdash E \ll F : \mathbf{SP}\ \tau\ \tau'}$$

Figure 8.3: Type rule for feed

8.1.2 Actions

The actions in our operational semantics represent the observations that we can make of Core Fudget terms. We consider three different kinds of action α as follows:

1. $\alpha = ?E$, an *input* action. A transition $, \vdash F : \tau \xrightarrow{?E} G$ means that F can evolve into G by receiving the term E on its input stream.

2. $\alpha = !E$, an *output* action. A transition $, \vdash F : \tau \xrightarrow{!E} G$ means that F can evolve into G by emitting the term E on its output stream.

3. $\alpha = \nu$, an *internal administrative* action. This action corresponds to the silent action τ of CCS. Here we use ν to avoid any confusion with the type τ. A transition $, \vdash F : \tau \xrightarrow{\nu} G$ means that F can evolve to G and requires no interaction with the external environment. Internal administrative actions arise from communications between stream processors.

Analogously to Core Fudget terms, we give typing rules for actions in Figure 8.4. The (*Red Act*) rule states that all internal administrative actions are well-typed. The rule for input actions associates the type of the term input with the input action. Similarly, in the case of output actions, the type of the term being output is assigned to the output action. It is important to type actions as their type is used to constrain the transitions that terms can make.

$$(\mathit{Red\ Act}) \quad , \vdash \nu : \tau \qquad (\mathit{Inp\ Act}) \quad \frac{, \vdash E : \tau}{, \vdash ?E : \tau} \qquad (\mathit{Out\ Act}) \quad \frac{, \vdash E : \tau}{, \vdash !E : \tau}$$

Figure 8.4: Type rules for actions

8.1.3 Transitions

We define the transition relation explicitly, as opposed to defining a reduction relation and a structural congruence relation in the style of Milner [Mil90, Mil91]. This simplifies proving properties of the transition relation as it is defined inductively by a set of inference rules for each possible form of Core Fudget term.

Stream processor transitions

Variables, constants and **NullSP** have no transitions, as they cannot evolve further. Terms built from **GetSP** and **PutSP** have transitions indicating their ability to perform input and output actions:

$$(\mathit{Inp}_1) \quad \frac{, \vdash ?F : \tau}{, \vdash \mathbf{GetSP}\ E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E\ F}$$

$$(\mathit{Out}) \quad , \vdash \mathbf{PutSP}\ E\ F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!E} F$$

Any terms comprising actions that are not mentioned in the left hand side of a transition are considered to be universally quantified. The (Inp_1) rule illustrates the need to extend the normal definition of transition with typing contexts. The hypothesis of this rule ensures that only terms corresponding to the type of the input stream τ can be read by the stream processor **GetSP** E. If we did not have this hypothesis then it would be possible for a transition to result in a term that is not well-typed.

Next, we consider the feed construct, which has two separate transition rules. The first rule allows a value to be fed into a stream processor that can perform an input transition of the appropriate term:

$$(\mathit{Feed}_1) \quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E'}{, \vdash E \ll F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu} E'}$$

The second transition rule for the feed construct captures the ability of output and administrative actions to occur in the term which is being fed values:

$$(Feed_2)\quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E'}{, \vdash E \ll F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E' \ll F}\quad \alpha \in \{\nu, !G\}$$

Serial composition has three separate transition rules, the first of these allows the rightmost stream processor to perform input and administrative transitions. The second rule supports the leftmost stream processor performing output and administrative transitions:

$$(Ser_1)\quad \frac{, \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} F'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\alpha} E >==< F'}\quad \alpha \in \{\nu, ?G\}$$

$$(Ser_2)\quad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{\alpha} E'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\alpha} E' >==< F}\quad \alpha \in \{\nu, !G\}$$

The final transition rule for serial composition corresponds to the communication of data between the two stream processors. This can occur when the rightmost stream processor can perform an output transition, and the value output is then fed into the leftmost stream processor using the feed construct:

$$(Ser_3)\quad \frac{, \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G} F'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu} (E \ll G) >==< F'}$$

Serial composition corresponds closely to parallel composition in a process calculi such as CCS. However, the (Ser_3) rule differs from the rule for synchronisation in CCS of a process that can perform an output with one that can perform an input. In particular, the leftmost stream processor in (Ser_3) does not need to be able to perform an input. The rule is modelling asynchronous output as the rightmost stream processor can perform its output action regardless of whether the leftmost stream processor is ready to consume input or not. Terms output from the rightmost stream processor are stored in the stream until the leftmost stream processor is ready to process them. The stream is thus modelled as a buffer using the feed construct, as successive outputs from the rightmost stream processor result in a chain of feed constructs into the leftmost stream processor. This allows the two stream processors to proceed at different rates.

We now turn our attention to the parallel composition of stream processors. Our first transition rule for parallel composition allows the leftmost stream processor to perform output and administrative transitions:

$$(Par_1)\quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E' >\!*\!< F} \quad \alpha \in \{\nu, !G\}$$

A symmetric version of this rule is also required, allowing the rightmost stream processor to perform output and administrative transition:

$$(Par_2)\quad \frac{, \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} F'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E >\!*\!< F'} \quad \alpha \in \{\nu, !G\}$$

All that remains is to consider when a parallel composition can perform an input transition. This case is quite subtle, and first we review the informal meaning of parallel composition. Values on the input stream to a parallel composition are received by *both* of the stream processors. This causes a problem if we attempt to define our transition rule similarly to the rule for parallel composition in CCS. The obvious transition rule is defined as:

$$\frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E' >\!*\!< F}$$

but this only allows one of the branches of the parallel composition to process the input value. The semantics we desire is for both of the branches to be able to process the input value. We can achieve this by buffering any input values on the branch not responsible for the input using the feed construct:

$$(Par_3)\quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E' >\!*\!< (F \ll G)}$$

A symmetric version of this rule is also required, allowing the rightmost stream processor to perform an input action:

$$(Par_4)\quad \frac{, \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} F'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} (E \ll G) >\!*\!< F'}$$

Next, we examine the feedback construct and its associated transition rules. The first rule allows the stream processor at the core of the feedback loop to perform internal administrative actions and inputs:

$$(Loop_1)\quad \frac{,\ \vdash E : \mathbf{SP}\ \tau\ \tau \xrightarrow{\alpha} E'}{,\ \vdash \mathbf{LoopSP}\ E : \mathbf{SP}\ \tau\ \tau \xrightarrow{\alpha} \mathbf{LoopSP}\ E'}\quad \alpha \in \{\nu, ?F\}$$

The second transition rule considers output produced by the the stream processor in the feedback loop. Any output produced must be emitted from the overall feedback loop, but also is fed back into the stream processor in the feedback loop using the feed construct:

$$(Loop_2)\quad \frac{,\ \vdash E : \mathbf{SP}\ \tau\ \tau \xrightarrow{!F} E'}{,\ \vdash \mathbf{LoopSP}\ E : \mathbf{SP}\ \tau\ \tau \xrightarrow{!F} \mathbf{LoopSP}\ (E' \ll F)}$$

This last rule for feedback gives the merging operation on the input side of the feedback loop a deterministic nature. Any values fed around the feedback loop are fed directly into the stream processor at the body of the loop. This corresponds to processing these values before any input pending on the overall input stream, which is the constraint discussed in Section 7.4. It is not clear how this rule could be modified to support a nondeterministic semantics for merging values on the input side of the feedback loop.

The remaining stream processor construct is the dynamic stream processor construct **DynSP**. First of all, we allow administrative and output transitions to occur underneath this construct:

$$(Dyn_1)\quad \frac{,\ \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E'}{,\ \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{\alpha} \mathbf{DynSP}\ E'}\quad \alpha \in \{\nu, !F\}$$

Next, we consider input transitions for dynamic stream processors. Any input term must first be reduced to a normal form, which in this case is either a left or right injection. This is necessary as the behaviour of the dynamic stream processor depends on whether the input is a left or right injection:

$$(Dyn_2)\quad \frac{,\ \vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \xrightarrow{\nu} G' \quad ,\ \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?G'} E'}{,\ \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?G} E'}$$

Once the input is in a normal form then the following transition rules analyse the input term and if it is tagged as a right injection then it is considered as normal input to the current stream processor embedded in the dynamic stream processor. Otherwise the input term must be a left injection and thus corresponds to a new stream processor that is used to replace the existing one:

$$(Dyn_3) \quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E'}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\mathbf{Right}\ G} \mathbf{DynSP}\ E'}$$

$$(Dyn_4) \quad \frac{, \vdash ?G : \mathbf{SP}\ \tau\ \tau'}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\mathbf{Left}\ G} \mathbf{DynSP}\ G}$$

The hypothesis in rule (Dyn_4) ensures that the current stream processor can only be replaced by another of the appropriate type.

Lambda calculus transitions

In this section we consider the transitions for the lambda calculus subset of the Core Fudgets language. There are two rules for function application. The first rule corresponds to β-reduction, whilst the second allows the reduction of the function in applications, corresponding to call-by-name evaluation. Other calling conventions can be easily simulated by using appropriate transitions:

$$(Lam_1) \quad , \vdash (\lambda x.E)\ F : \tau \xrightarrow{\nu} E[F/x]$$

$$(Lam_2) \quad \frac{, \vdash E : \tau \to \tau' \xrightarrow{\nu} E'}{, \vdash E\ F : \tau' \xrightarrow{\nu} E'\ F}$$

Recursion is unwound by substituting the recursive term for the recursive variable as necessary. If a transition is possible when we unwind the recursion by one level then the overall recursive term can also make the same transition:

$$(Rec) \quad \frac{, \vdash E[(\mathbf{Fix}\ x.E)/x] : \tau \xrightarrow{\alpha} E'}{, \vdash \mathbf{Fix}\ x.E : \tau \xrightarrow{\alpha} E'}$$

Constant function transitions

This section describes the transition rules for constant functions, and we begin by considering the rules to analyse sum types. Initially the term being analysed must be reduced until it is either a left or right injection. Once this has happened then its value can be used to determine the behaviour of the overall analysis. This is captured by the following rules:

$$(Case_1)\quad \frac{, \vdash E : \tau + \tau' \xrightarrow{\nu} E'}{, \vdash \mathbf{Case}\ E \to F, G : \tau'' \xrightarrow{\nu} \mathbf{Case}\ E' \to F, G}$$

$$(Case_2)\quad , \vdash \mathbf{Case\ Left}\ E \to F, G : \tau'' \xrightarrow{\nu} F\ E$$

$$(Case_3)\quad , \vdash \mathbf{Case\ Right}\ E \to F, G : \tau'' \xrightarrow{\nu} G\ E$$

Conditionals are treated in a similar manner by first reducing the boolean being tested to either **true** or **false**. This value is then used to determine the behaviour of the conditional construct:

$$(Cond_1)\quad \frac{, \vdash E : \mathbf{bool} \xrightarrow{\nu} E'}{, \vdash \mathbf{if}\ E\ \mathbf{then}\ F\ \mathbf{else}\ G : \tau \xrightarrow{\nu} \mathbf{if}\ E'\ \mathbf{then}\ F\ \mathbf{else}\ G}$$

$$(Cond_2)\quad , \vdash \mathbf{if\ true\ then}\ E\ \mathbf{else}\ F : \tau \xrightarrow{\nu} E$$

$$(Cond_3)\quad , \vdash \mathbf{if\ false\ then}\ E\ \mathbf{else}\ F : \tau \xrightarrow{\nu} F$$

The addition of natural numbers follows a similar pattern, by first reducing the two arguments to the operator until they are both numbers and then performing the addition. The following rules capture this mechanism:

$$(Add_1)\quad \frac{, \vdash E : \mathbf{num} \xrightarrow{\nu} E'}{, \vdash \mathbf{add}\ E\ F : \mathbf{num} \xrightarrow{\nu} \mathbf{add}\ E'\ F}$$

$$(Add_2)\quad \frac{, \vdash F : \mathbf{num} \xrightarrow{\nu} F'}{, \vdash \mathbf{add}\ E\ F : \mathbf{num} \xrightarrow{\nu} \mathbf{add}\ E\ F'}$$

$$(Add_3)\quad , \vdash \mathbf{add\ n\ m} : \mathbf{num} \xrightarrow{\nu} \mathbf{n} + \mathbf{m}$$

Finally, the comparison of natural numbers for equality proceeds by first reducing the two arguments to the operator until they are both numbers and then performing the actual comparison. The following rules capture this mechanism:

$$(Equal_1)\quad \frac{, \vdash E : \mathbf{num} \xrightarrow{\nu} E'}{, \vdash \mathbf{equal}\ E\ F : \mathbf{bool} \xrightarrow{\nu} \mathbf{equal}\ E'\ F}$$

$$(Equal_2)\quad \frac{, \vdash F : \mathbf{num} \xrightarrow{\nu} F'}{, \vdash \mathbf{equal}\ E\ F : \mathbf{bool} \xrightarrow{\nu} \mathbf{equal}\ E\ F'}$$

$$(Equal_3)\qquad , \vdash \mathbf{equal\ n\ m} : \mathbf{bool} \xrightarrow{\nu} \mathbf{n} = \mathbf{m}$$

Subject reduction

In this section we prove some properties of the transition relation. We show that, if a term can perform an output action, then the type of the action must correspond to the type of the output stream of the term. A similar result holds for input actions, and we use both of these results to prove a subject reduction property:

Lemma 8.1 (Subject reduction)

1. *if* $, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!F} E'$ *then* $, \vdash F : \tau'$;
2. *if* $, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E'$ *then* $, \vdash F : \tau$;
3. *if* $, \vdash E : \tau \xrightarrow{\alpha} E'$ *then* $, \vdash E' : \tau$.

Proof. *Parts (1) and (2) result from a simple induction on the inference of the transitions* $, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!F} E'$, *and* $, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E'$, *respectively. Part (3) follows by induction on the inference of* $, \vdash E : \tau \xrightarrow{\alpha} E'$ *and by using Lemma 4.1, and parts (1) and (2). We examine one of the cases for part (3):*
Case (Par_3). *We have the derivation:*

$$\frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E' >\!*\!< (F \ll G)}$$

By induction we have that $, \vdash E' : \mathbf{SP}\ \tau\ \tau'$, *and from part (2) we can show that* $, \vdash G : \tau$. *Using the (Feed) transition rule gives* $, \vdash F \ll G : \mathbf{SP}\ \tau\ \tau'$ *and from the (Parallel) rule we obtain* $, \vdash E' >\!*\!< (F \ll G) : \mathbf{SP}\ \tau\ \tau'$.

The other cases follow in a similar manner. ∎

8.1.4 Examples

Before we consider formalising a theory for Core Fudgets, we pause for some examples, illustrating the operational semantics and its nondeterministic nature. The first example is given in Figure 8.5 and illustrates a stream processor producing two constants on its output stream. For simplicity we have omitted the typing context, and the source type of the transitions.

Figure 8.6 shows the second example, a stream processor that discards the second value it is sent, but forwards the first value sent to it. There is an infinite family of initial transitions, one for each possible instantiation of the input variable x. The only constraint being that the possible instantiations must have an appropriate type corresponding to the input stream of the overall stream processor. We only show one of the possible transitions for an arbitrary instantiation of x, and take this approach in the remaining examples too.

As an application of recursion, Figure 8.7, shows the meaning attributed to the identity stream processor. The initial transition is due to the body of the recursive definition and corresponds to unwinding the definition one level. This example clearly illustrates that any terms received as input by the identity stream processor must be output before any more input terms can be processed.

The last example, in Figure 8.8, shows the nondeterministic nature of the semantics for serial composition of stream processors. The initial term has two possible transitions as the term can either perform an output or read some input.

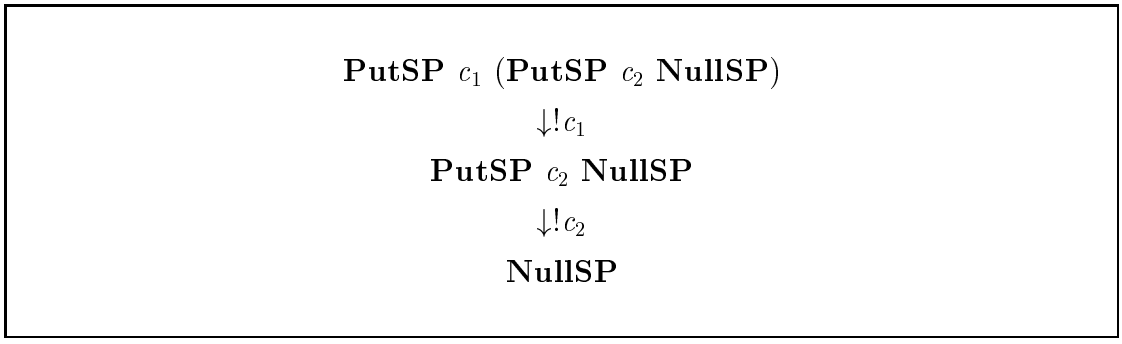


Figure 8.5: An output example

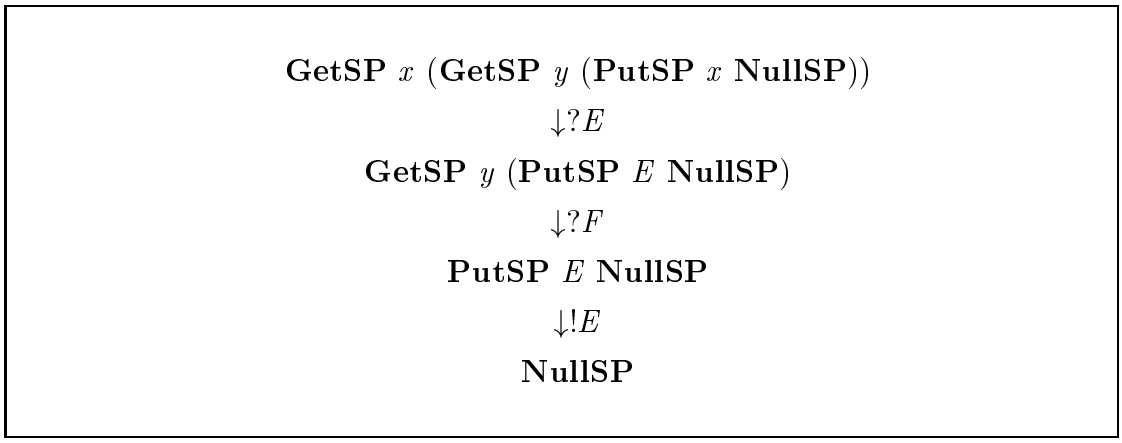


Figure 8.6: The discard example

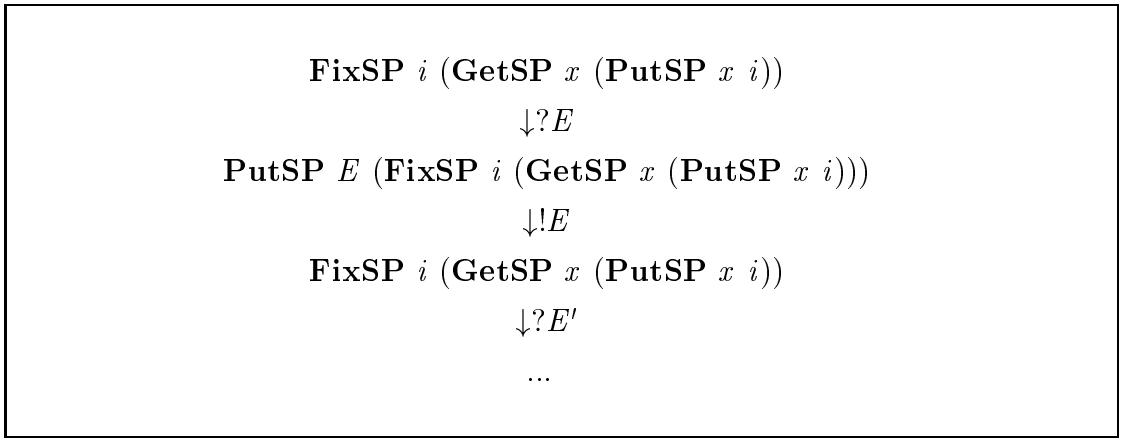


Figure 8.7: The semantics of the identity stream processor

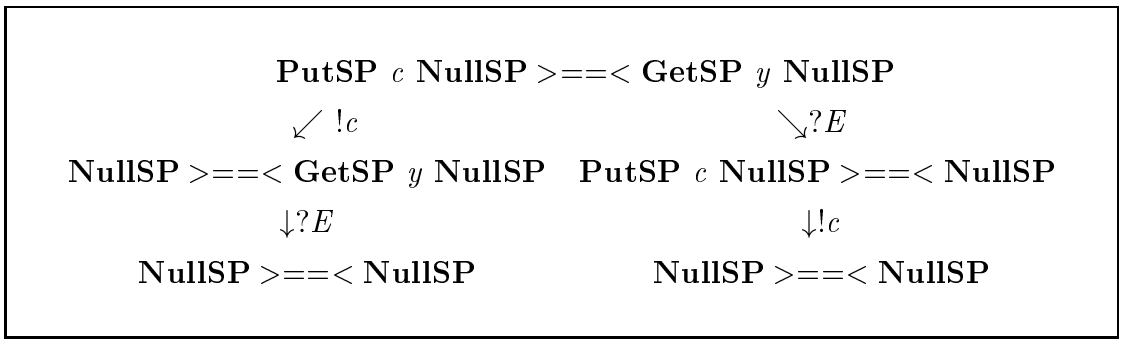


Figure 8.8: A mixed input and output example

8.2 A semantic theory

We now focus our attention on defining an appropriate equivalence for Core Fudget terms. A higher-order variation of weak early bisimulation equivalence is the one that we finally choose. Intuitively, this compares terms of the language ignoring internal administrative actions, and comparing the terms sent along streams using the variation of weak early bisimulation equivalence itself.

We require our equivalences only to relate terms of the same type. This can be formalised by defining the equivalences as type-indexed families of relations. A type indexed relation is analogous to the confined relations introduced in Chapter 7 for the π-calculus:

Definition 8.1 (Type-indexed relation) *A family of relations $\mathcal{R}_{\Gamma,\tau}$ between terms of the same type τ and context Γ, is a type-indexed relation if $E\,\mathcal{R}_{\Gamma,\tau}\,F$ implies that $\Gamma \vdash E : \tau$ and $\Gamma \vdash F : \tau$. We write $\Gamma \vdash E\,\mathcal{R}\,F : \tau$ to mean that $E\,\mathcal{R}_{\Gamma,\tau}\,F$, and usually omit any obvious type information.*

A type-indexed family of relations is *closed* when each of the indexed relations in the family is closed. Initially, we consider equivalences on closed terms only, and thus formulate them as closed type-indexed relations. These equivalences are extended to open terms by considering all closing substitutions:

Definition 8.2 (Open extension) *For an arbitrary typing context Γ, such that $\Gamma = \vec{x} : \vec{\tau}$, then a Γ-Closure is a substitution $\cdot[\vec{G}/\vec{x}]$ where $\emptyset \vdash \vec{G} : \vec{\tau}$. The open extension $\mathcal{R}^\circ$ of a closed type-indexed relation $\mathcal{R}$ is the least type-indexed relation such that:*

- *$\Gamma \vdash E\,\mathcal{R}^\circ\,F : \tau$ iff for all Γ-closures $\cdot[\vec{G}/\vec{x}]$ then $\emptyset \vdash E[\vec{G}/\vec{x}]\,\mathcal{R}\,F[\vec{G}/\vec{x}] : \tau$.*

To form the basis of a useful semantic theory an equivalence must be preserved by all of the constructs in the language. An equivalence relation with this property is a congruence, and we capture this formally by using contexts:

Definition 8.3 (Context) *A context is an expression containing a single hole. We use the metavariable $\mathcal{C}$ to range over contexts, and denote them as $\vdash \mathcal{C}[-_\tau] : \tau'$, where the hole satisfies the typing judgement $\vdash -_\tau : \tau$. The instantiation of a context with a term E of the appropriate type, $\mathcal{C}[E]$, corresponds to filling the hole in $\mathcal{C}$ with the term E, which may capture variables free in E.*

Intuitively, an equivalence relation is a congruence if, for any two terms that are related by the equivalence, then the terms resulting from substituting these two terms into any context are also equivalent:

Definition 8.4 (Congruence$_1$) *A type-indexed equivalence relation, $\mathcal{R}$, is a congruence if it satisfies the following inference rule:*

$$\textit{(Cong R)} \quad \frac{, \vdash \mathcal{C}[-_\tau] : \tau' \quad , , , ' \vdash E \mathcal{R} E' : \tau}{, \vdash \mathcal{C}[E] \mathcal{R} \mathcal{C}[E'] : \tau'}$$

where , ' is the list of bound variables in the context $\mathcal{C}$.

This definition corresponds directly to the intuition required for an equivalence to be useful for equational reasoning. It states that, if two terms are related by the equivalence, then there is no context that can tell the two terms apart, and so we are free to replace one term by the other.

Before we consider various forms of bisimulation for Core Fudgets, we introduce a structural property capturing the equivalence of normal forms. These have to be treated separately as they have no transitions under the operational semantics, and so bisimulation would equate all of them. We define a class of structure-preserving relations that identify normal forms in an appropriate manner as follows:

Definition 8.5 (Structure-preserving) *A type-indexed relation $\mathcal{R}$ is structure-preserving if all the following requirements are satisfied, where $\Longrightarrow$ is the transitive and reflexive closure of $\xrightarrow{\nu}$:*

- *if* $, \vdash L \mathcal{R} E : \tau$ *and* $\tau \in \{\mathbf{bool}, \mathbf{num}\}$ *then* $, \vdash E : \tau \Longrightarrow M$ *and* $L = M$*;*
- *if* $, \vdash \mathbf{Left}\ E \mathcal{R} F : \tau + \tau'$ *then* $, \vdash F : \tau + \tau' \Longrightarrow \mathbf{Left}\ F'$ *and* $, \vdash E \mathcal{R} F' : \tau$*;*
- *if* $, \vdash \mathbf{Right}\ E \mathcal{R} F : \tau + \tau'$ *then* $, \vdash F : \tau + \tau' \Longrightarrow \mathbf{Right}\ F'$ *and* $, \vdash E \mathcal{R} F' : \tau'$*;*
- *if* $, \vdash (\lambda x.E) \mathcal{R} F : \tau \to \tau'$ *then* $, \vdash F : \tau \to \tau' \Longrightarrow (\lambda x.F')$ *and for all terms* $, \vdash G : \tau$ *we must have* $, \vdash ((\lambda x.E)\ G) \mathcal{R} ((\lambda x.F')\ G) : \tau'$.

An alternative approach to dealing with normal forms is to extend the labelled transition system of the operational semantics to make normal forms observable. Here we have opted not to take this approach as we are concerned primarily with the observations of stream processors.

8.2.1 First-order bisimulation

Our starting point for an analysis of appropriate equivalences for Core Fudgets is *strong first-order* bisimulation. We extend the usual definition of this equivalence to typed-indexed relations as follows:

Definition 8.6 (Strong first-order simulation) *$\mathcal{R}$ is a strong first-order simulation if it is a closed type-indexed family of relations that are structure-preserving and $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *whenever $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ then $\exists F'.\emptyset \vdash F : \tau \xrightarrow{\alpha} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form (for simplicity types have been omitted):

$$\begin{array}{ccc} E & \mathcal{R} & F \\ {\scriptstyle\alpha}\downarrow & & \\ E' & & \end{array} \qquad as \qquad \begin{array}{ccc} E & \mathcal{R} & F \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\alpha} \\ E' & \mathcal{R} & F' \end{array}$$

Definition 8.7 (Strong first-order bisimulation) *$\mathcal{R}$ is a strong first-order bisimulation if it is a strong first-order simulation and also symmetric. Strong first-order bisimilarity $\sim^1$ is defined to be the greatest such relation.*

Proposition 8.1 *$\sim^1$ is an equivalence.*

Proof. *By definition $\sim^1$ is symmetric, and it is straightforward to show the relation $\{(E, E) \mid , \ \vdash E \sim^1 E : \tau\}$ to be a strong first-order bisimulation thus proving reflexivity. Similarly, we prove transitivity by showing that $\sim^1 \circ \sim^1$ is a strong first-order bisimulation.* ∎

This equivalence is *strong* as it requires internal administrative ν actions to be matched exactly. This yields a rather strict relation, and it is often the case that we want to abstract away from the internal behaviour of terms. This corresponds to transitions of the form $E \xrightarrow{\nu} F$ being unobservable. For example, we would intuitively expect the terms in Figure 8.9 to be equivalent because both have an output transition. However, B must first make an internal administrative ν transition before it can make this output transition, and thus $A \not\sim^1 B$.

$$
\begin{array}{lcl}
A & = & \textbf{PutSP } c \textbf{ NullSP} \\
B & = & \textbf{GetSP } (\lambda x.\textbf{PutSP } c \textbf{ NullSP}) >==< \textbf{PutSP } c \textbf{ NullSP}
\end{array}
$$

Figure 8.9: An example of terms not $\sim^1$ equivalent

This problem can be solved by using the weaker form of bisimulation, referred to as *weak bisimulation*, which doesn't match ν transitions exactly. First, we define weak transitions which are used in the definition of weak bisimulation:

Definition 8.8 (Weak transition) *Let $\Longrightarrow$ be the reflexive and transitive closure of $\stackrel{\nu}{\longrightarrow}$, and $\stackrel{\alpha}{\Longrightarrow}$ be the relational composition $\Longrightarrow \circ \stackrel{\alpha}{\longrightarrow} \circ \Longrightarrow$. A weak transition, $\stackrel{\widehat{\alpha}}{\Longrightarrow}$ is defined as:*

$$\stackrel{\widehat{\alpha}}{\Longrightarrow} = \begin{cases} \stackrel{\alpha}{\Longrightarrow} & \alpha \neq \nu \\ \Longrightarrow & \textit{otherwise.} \end{cases}$$

Definition 8.9 (Weak first-order simulation) *$\mathcal{R}$ is a weak first-order simulation if it is a closed type-indexed family of relations that are structure-preserving and $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *whenever $\emptyset \vdash E : \tau \stackrel{\alpha}{\longrightarrow} E'$ then $\exists F'.\emptyset \vdash F : \tau \stackrel{\widehat{\alpha}}{\Longrightarrow} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$.*

intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form (for simplicity types have been omitted):

$$\begin{array}{ccc} E & - \mathcal{R} - & F \\ {\scriptstyle\alpha}\downarrow & & \\ E' & & \end{array} \qquad \textit{as} \qquad \begin{array}{ccc} E & - \mathcal{R} - & F \\ {\scriptstyle\alpha}\downarrow & & \Downarrow{\scriptstyle\widehat{\alpha}} \\ E' & - \mathcal{R} - & F' \end{array}$$

Definition 8.10 (Weak first-order bisimulation) *$\mathcal{R}$ is a weak first-order bisimulation if it is a weak first-order simulation and also symmetric. Weak first-order bisimilarity $\approx^1$ is defined to be the greatest such relation.*

Proposition 8.2 *$\approx^1$ is an equivalence.*

Proof. *The proof follows in a similar manner to Proposition 8.1.* ∎

Both strong and weak first-order bisimilarity are not preserved by all of the constructs in the Core Fudgets language. For example, although $\mathbf{0} + \mathbf{1} \approx^1 \mathbf{1} + \mathbf{0}$, this equivalence is not preserved by **PutSP** because **PutSP** $(\mathbf{0} + \mathbf{1})$ **NullSP** $\not\approx^1$ **PutSP** $(\mathbf{1} + \mathbf{0})$ **NullSP**. This counterexample also holds for strong first-order bisimilarity. This makes both equivalences too strict to form the basis of a useful equational theory for Core Fudgets.

8.2.2 Higher-order bisimulation

The first-order matching of actions results in equivalences that are too fine, as terms such as **PutSP** $(E >\!*\!< F)$ **NullSP** and **PutSP** $(F >\!*\!< E)$ **NullSP** are distinguishable. Here, we follow the solution taken by Thomsen [Tho90], based on earlier work by Astesiano [AGR88] and Boudol [Bou89], where the terms output in an output transition must be bisimilar rather than identical. This form of bisimulation is called *higher-order bisimulation*, and we formalise it by first extending type-indexed relations to actions:

Definition 8.11 (Action extension) *A type-indexed relation $\mathcal{R}$ can be extended to a relation over actions $\mathcal{R}^a$, by the inductive rules listed in Figure 8.10, which match output actions using the relation $\mathcal{R}$.*

$$\frac{}{, \vdash \nu \mathcal{R}^a \nu : \tau} \qquad \frac{}{, \vdash ?E \mathcal{R}^a ?E : \tau} \qquad \frac{, \vdash E \,\mathcal{R}\, F : \tau}{, \vdash !E \mathcal{R}^a !F : \tau}$$

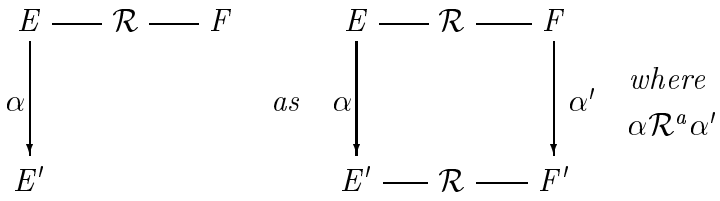
Figure 8.10: Action extension rules

Strong higher-order simulation is defined similarly to its first-order counterpart:

Definition 8.12 (Strong higher-order simulation) *$\mathcal{R}$ is a strong higher-order simulation if it is a closed type-indexed family of relations that are structure-preserving and $\emptyset \vdash E \mathcal{R} F : \tau$ implies:*

- *whenever $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ then $\exists F', \alpha'. \emptyset \vdash F : \tau \xrightarrow{\alpha'} F'$ and $\emptyset \vdash E' \mathcal{R} F' : \tau$ and $\emptyset \vdash \alpha \,\mathcal{R}^a\, \alpha' : \tau'$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form (for simplicity types have been omitted):



Definition 8.13 (Strong higher-order bisimulation) *A strong higher-order simulation $\mathcal{R}$ is a strong-higher order bisimulation if it is symmetric. Strong higher-order bisimilarity $\sim^h$ is defined to be the greatest such relation.*

Proposition 8.3 *$\sim^h$ is an equivalence.*

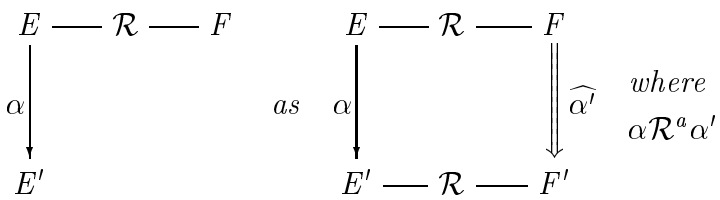
Proof. *The proof follows in a similar manner to Proposition 8.1.* ∎

Strong higher-order bisimulation is not particularly useful as the abundance of ν transitions in the operational semantics stops us from showing even the simplest of properties. For example, $(\mathbf{0}+\mathbf{0})+\mathbf{0} \not\sim^h \mathbf{0}+\mathbf{0}$ because the left hand side has two ν transitions while the right hand side has only one. However, the natural extension to use weak transitions, called weak higher-order bisimulation, is a much more interesting equivalence that allows us to prove such properties:

Definition 8.14 (Weak higher-order simulation) *$\mathcal{R}$ is a weak higher-order simulation if it is a closed type-indexed family of relations that are structure-preserving and $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *whenever $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ then $\exists F', \alpha'.\emptyset \vdash F : \tau \stackrel{\widehat{\alpha'}}{\Longrightarrow} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$ and $\emptyset \vdash \alpha\ \mathcal{R}^a\ \alpha' : \tau'$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form (for simplicity types have been omitted):



Definition 8.15 (Weak higher-order bisimulation) *$\mathcal{R}$ is a weak higher-order bisimulation if it is a weak higher-order simulation and also symmetric. Weak higher-order bisimilarity $\approx^h$ is defined to be the greatest such relation.*

Proposition 8.4 *$\approx^h$ is an equivalence.*

Proof. *The proof follows in a similar manner to Proposition 8.1.* ∎

This equivalence is our final candidate for the basis of an equational theory for Core Fudgets. As such it should be the case that $\approx^h$ is preserved by all of the constructs of the language. This can be shown to be the case by proving $\approx^{h^\circ}$ to be a congruence and hence it must also be that $\approx^h$ is a congruence:

Theorem 8.1 (Congruence) *$\approx^{h^\circ}$ is a congruence.*

Proof. *It is easy to show that $\approx^{h^\circ}$ is preserved by the input and output constructs,* **GetSP***, and* **PutSP***, respectively. However, it is much more difficult to show that $\approx^{h^\circ}$ is preserved by serial composition of stream processors:*

- *if* , $\vdash E_1 \approx^{h^\circ} E_2 : \mathbf{SP}\ \tau'\ \tau''$ *and* , $\vdash F_1 \approx^{h^\circ} F_2 : \mathbf{SP}\ \tau\ \tau'$ *then* , $\vdash E_1 >==< F_1 \approx^{h^\circ} E_2 >==< F_2 : \mathbf{SP}\ \tau\ \tau''$.

Consider the terms:

$$E_1 = \mathbf{GetSP}\ (\lambda x.G_1) \qquad E_2 = \mathbf{GetSP}\ (\lambda x.G_2)$$
$$F_1 = \mathbf{PutSP}\ V_1\ \mathbf{NullSP} \quad F_2 = \mathbf{PutSP}\ V_2\ \mathbf{NullSP}$$

with , $\vdash G_1 \approx^{h^\circ} G_2 : \tau' \to \mathbf{SP}\ \tau'\ \tau''$ *and* , $\vdash V_1 \approx^{h^\circ} V_2 : \tau'$*, then in order to show that* , $\vdash E_1 >==< F_1 \approx^{h^\circ} E_2 >==< F_2 : \mathbf{SP}\ \tau\ \tau''$ *requires proving that* , $\vdash G_1[V_1/x] \approx^{h^\circ} G_2[V_2/x] : \mathbf{SP}\ \tau'\ \tau''$*. The full proof follows by a typed reworking of Howe's method [How89], and is detailed in Appendix A.* ∎

This is our coarsest equivalence, and we take it as the basis of our semantic theory. When we present equivalences between Core Fudget terms we use the finest applicable equivalence, although it is always reasonable to just use $\approx^h$. Figure 8.11 shows the relationship among the equivalences, with an arrow meaning inclusion.

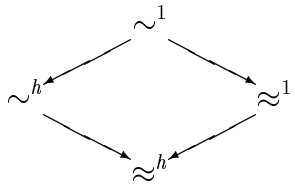


Figure 8.11: The relationship between the bisimulation equivalences

8.2.3 Examples

We conclude by illustrating the semantic theory with some examples. In the next chapter we use this theory to develop a set of equational rules, and so here we only show that some standard rules of the λ-calculus hold for Core Fudgets.

Firstly, we show that the semantic equivalence captures β-equivalence as expected. More formally, $\emptyset \vdash (\lambda x.E)\ F \approx^1 E[F/x] : \tau$. In order to prove this we must consider all the possible transitions of the first term, and find a corresponding weak transition of the second term:

$$\begin{array}{ccc} \emptyset \vdash (\lambda x.E)\ F & \approx^1 & E[F/x] : \tau \\ \Big\downarrow \nu & & \Big\Downarrow \\ \emptyset \vdash E[F/x] & \approx^1 & E[F/x] : \tau \end{array}$$

The transition of the first term is due to the (Lam_1) rule, while the matching weak transition of the second term comes from the reflexivity of $\stackrel{\widehat{\nu}}{\Longrightarrow}$. The reflexivity of $\approx^1$ gives us $E[F/x] \approx^1 E[F/x]$. For the symmetric argument, we need to consider the possible transitions of $E[F/x]$ and find matching weak transitions of the term $(\lambda x.E)\ F$, which follows as $(\lambda x.E)\ F$ has the above ν transition to $E[F/x]$.

Similarly, it is straightforward to show that η-equivalence is also captured by the semantic theory. Formally, we are interested in showing that the equation $\emptyset \vdash \lambda x.(E\ x) \approx^1 E : \tau \to \tau'$ holds, where x does not occur free in E. As both of the terms $\lambda x.(E\ x)$ and E have function types then the structure-preserving nature of weak first-order bisimulation requires that:

- $\forall W.\emptyset \vdash W : \tau$ implies $\emptyset \vdash (\lambda x.(E\ x))\ W \approx^1 E\ W : \tau'$.

However, there is only one transition of the first term due to the (Lam_1) rule:

$$\begin{array}{ccc} \emptyset \vdash (\lambda x.(E\ x))\ W & \approx^1 & E\ W : \tau' \\ \Big\downarrow \nu & & \Big\Downarrow \\ \emptyset \vdash (E\ x)[W/x] = E\ W & \approx^1 & E\ W : \tau' \end{array}$$

The matching weak transition is due to the reflexivity of $\stackrel{\widehat{\nu}}{\Longrightarrow}$, and $E\ W \approx^1 E\ W$ because $\approx^1$ is reflexive. The symmetric argument follows in a similar manner to that for β-equivalence.

Chapter 9

Axioms and applications

This chapter explores the properties the operational semantics of the last chapter bestows upon the Core Fudgets language. In particular, a set of axioms are developed and are shown to be correct according to the semantic equivalence of Chapter 8. More interesting are some axioms that one might expect to be correct but which prove to be incorrect according to the operational semantics. We discuss some of these axioms and the reason for why they do not hold in our model.

The operational semantics is also useful for assessing the correctness of implementations of the Core Fudgets language. We consider an implementation to be correct if it respects linear time and is also sensitive to termination. This formulation of correctness is illustrated by considering the original Chalmers implementation of the Fudgets system. We find that this implementation does not quite meet our requirements for it to be correct, but show how a simple change can solve this problem.

9.1 Axioms

Although the semantic theory for the operational semantics is more direct than using the π-calculus encoding from Chapter 6, it is practically more useful to develop equational rules from the theory. These rules or axioms can then be used to reason about Core Fudget programs without recourse to the technicalities of the semantic theory. In this section we examine a set of such axioms, and prove them to be correct using the semantic equivalence defined in the previous chapter.

To begin, there are the obvious rules establishing **NullSP** as a zero for the feed construct, and also showing that the feed construct distributes over parallel and serial composition of stream processors. Although these rules are intuitively correct, we are now in a position to prove them formally, using the semantic theory from Chapter 8. We note that the feed construct is *not* associative, because $(E \ll F) \ll G$ corresponds to feeding the term F into E and then feeding the term G into the resulting stream processor. However, $E \ll (F \ll G)$ is not the same, as it corresponds to feeding the term $F \ll G$ into E.

Theorem 9.1 (Feed axioms)

$$\begin{array}{rcll} \mathbf{NullSP} \ll E & \sim^1 & \mathbf{NullSP} & (\mathit{Zero}) \\ (E >\!*\!< F) \ll G & \sim^1 & (E \ll G) >\!*\!< (F \ll G) & (\mathit{Distributivity}_1) \\ (E >\!=\!=\!< F) \ll G & \sim^1 & E >\!=\!=\!< (F \ll G) & (\mathit{Distributivity}_2) \end{array}$$

Proof. *It is straightforward to show that the relations:*

$$\begin{array}{rcl} \mathcal{S}_a & = & \{(\mathbf{NullSP} \ll E,\ \mathbf{NullSP}) \mid \emptyset \vdash E : \tau\} \\ \mathcal{S}_{d1} & = & \{((E >\!*\!< F) \ll G,\ (E \ll G) >\!*\!< (F \ll G)) \\ & & \quad \mid \emptyset \vdash E : \mathbf{SP}\ \tau\ \tau' \wedge \emptyset \vdash F : \mathbf{SP}\ \tau\ \tau' \wedge \emptyset \vdash G : \tau\} \cup \sim^1 \\ \mathcal{S}_{d2} & = & \{((E >\!=\!=\!< F) \ll G,\ E >\!=\!=\!< (F \ll G)) \\ & & \quad \mid \emptyset \vdash E : \mathbf{SP}\ \tau'\ \tau'' \wedge \emptyset \vdash F : \mathbf{SP}\ \tau\ \tau' \wedge \emptyset \vdash G : \tau\} \end{array}$$

are strong first-order bisimulations. We examine the second relation $\mathcal{S}_{d1}$ in detail, with the others following in a similar manner. Consider any a and b where $\emptyset \vdash a \mathcal{S}_{d1} b : \tau$, then we must check that each transition $\emptyset \vdash a \xrightarrow{\alpha} a' : \tau$ is matched by a transition $\emptyset \vdash b \xrightarrow{\alpha} b' : \tau$ such that $\emptyset \vdash a' \mathcal{S}_{d1} b' : \tau$. The symmetric case for each transition $\emptyset \vdash b \xrightarrow{\alpha} b' : \tau$ must also be checked accordingly. Let a be the term $\emptyset \vdash (E >\!*\!< F) \ll G : \mathbf{SP}\ \tau\ \tau'$ and b be $\emptyset \vdash (E \ll G) >\!*\!< (F \ll G) : \mathbf{SP}\ \tau\ \tau'$ and consider the possible transitions of E, which are also the same for F:

Case $\emptyset \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu} E'$. *Then we have matching transitions for a and b*

$$\begin{array}{rcl} \emptyset \vdash a : \mathbf{SP}\ \tau\ \tau' & \xrightarrow{\nu} & (E' >\!*\!< F) \ll G \\ \emptyset \vdash b : \mathbf{SP}\ \tau\ \tau' & \xrightarrow{\nu} & (E' \ll G) >\!*\!< (F \ll G) \end{array}$$

where $\emptyset \vdash (E' >\!*\!< F) \ll G\ \mathcal{S}_{d1}\ (E' \ll G) >\!*\!< (F \ll G) : \mathbf{SP}\ \tau\ \tau'$;

Case $\emptyset \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E'$. *We have the corresponding transitions:*

$$\begin{array}{rcl} \emptyset \vdash a : \mathbf{SP}\ \tau\ \tau' & \xrightarrow{\nu} & E' >\!\!*\!\!< (F \ll G) \\ \emptyset \vdash b : \mathbf{SP}\ \tau\ \tau' & \xrightarrow{\nu} & E' >\!\!*\!\!< (F \ll G) \end{array}$$

where $\emptyset \vdash E' >\!\!*\!\!< (F \ll G) \sim^1 E' >\!\!*\!\!< (F \ll G) : \mathbf{SP}\ \tau\ \tau'$ *due to the reflexivity of* $\sim^1$ *and thus the residual terms are related by* $\mathcal{S}_{d1}$ *as required,* $\emptyset \vdash E' >\!\!*\!\!< (F \ll G) \mathcal{S}_{d1} E' >\!\!*\!\!< (F \ll G) : \mathbf{SP}\ \tau\ \tau'$*;*

Case $\emptyset \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!H} E'$. *Then we can deduce the transitions:*

$$\begin{array}{rcl} \emptyset \vdash a : \mathbf{SP}\ \tau\ \tau' & \xrightarrow{!H} & (E' >\!\!*\!\!< F) \ll G \\ \emptyset \vdash b : \mathbf{SP}\ \tau\ \tau' & \xrightarrow{!H} & (E' \ll G) >\!\!*\!\!< (F \ll G) \end{array}$$

where $\emptyset \vdash (E' >\!\!*\!\!< F) \ll G\ \mathcal{S}_{d1}\ (E' \ll G) >\!\!*\!\!< (F \ll G) : \mathbf{SP}\ \tau\ \tau'$.

■

Next, we consider the rules for parallel composition which show **NullSP** to be an identity, and that parallel composition is commutative and associative. However, parallel composition is *not* idempotent, because if $\emptyset \vdash E \xrightarrow{!F} E' : \mathbf{SP}\ \tau\ \tau'$ then , $\vdash E >\!\!*\!\!< E : \mathbf{SP}\ \tau\ \tau'$ has two output actions unlike E which has only one.

Theorem 9.2 (Parallel axioms)

$$\begin{array}{rcll} \mathbf{NullSP} >\!\!*\!\!< E & \sim^1 & E & (\textit{Identity}) \\ E >\!\!*\!\!< F & \sim^1 & F >\!\!*\!\!< E & (\textit{Commutativity}) \\ (E >\!\!*\!\!< F) >\!\!*\!\!< G & \sim^1 & E >\!\!*\!\!< (F >\!\!*\!\!< G) & (\textit{Associativity}) \end{array}$$

Proof. *For the following relations:*

$$\begin{array}{rcl} \mathcal{S}_i & = & \{(\mathbf{NullSP} >\!\!*\!\!< E, E) \mid \emptyset \vdash E : \mathbf{SP}\ \tau\ \tau'\} \\ \mathcal{S}_c & = & \{(E >\!\!*\!\!< F, F >\!\!*\!\!< E) \mid \emptyset \vdash E : \mathbf{SP}\ \tau\ \tau' \wedge \emptyset \vdash F : \mathbf{SP}\ \tau\ \tau'\} \\ \mathcal{S}_a & = & \{((E >\!\!*\!\!< F) >\!\!*\!\!< G, E >\!\!*\!\!< (F >\!\!*\!\!< G)) \\ & & \quad \mid \emptyset \vdash E : \mathbf{SP}\ \tau\ \tau' \wedge \emptyset \vdash F : \mathbf{SP}\ \tau\ \tau' \wedge \emptyset \vdash G : \mathbf{SP}\ \tau\ \tau'\} \end{array}$$

it is straightforward to show $\mathcal{S}_c$ *to be a strong first-order bisimulation. Following the standard technique of bisimulation up to [Mil85], and by using the fact that* $\sim^1$ *is preserved by parallel composition of stream processors, it is also straightforward to show that* $\mathcal{S}_i$ *and* $\mathcal{S}_a$ *are strong first-order bisimulations up to* $\sim^1$. ■

Serial composition of stream processors is associative, but **NullSP** is not a general identity for it. The problem is that in the term $\mathbf{NullSP} >==< E$ the stream processor E may be able to consume input, unlike **NullSP**. Similarly, the term $E >==< \mathbf{NullSP}$ may be able to produce output if E can produce output unlike **NullSP**. We prove one restricted identity rule, stating that a stream processor waiting for input serially composed with **NullSP** is equivalent to **NullSP**.

Theorem 9.3 (Serial axioms)

$$\begin{array}{rcll} \mathbf{GetSP}\ E >==< \mathbf{NullSP} & \sim^1 & \mathbf{NullSP} & (\mathit{Identity}_1) \\ (E >==< F) >==< G & \sim^1 & E >==< (F >==< G) & (\mathit{Associativity}) \end{array}$$

Proof. *For the following relations:*

$$\begin{array}{rcl} \mathcal{S}_{i1} & = & \{(\mathbf{GetSP}\ E >==< \mathbf{NullSP}, \mathbf{NullSP}) \mid \emptyset \vdash E : \tau \to \mathbf{SP}\ \tau\ \tau'\} \\ \mathcal{S}_a & = & \{((E >==< F) >==< G, E >==< (F >==< G)) \\ & & \quad \mid \emptyset \vdash E : \mathbf{SP}\ \tau''\ \tau''' \wedge \emptyset \vdash F : \mathbf{SP}\ \tau'\ \tau'' \wedge \emptyset \vdash G : \mathbf{SP}\ \tau\ \tau'\} \end{array}$$

it is straightforward to establish $\mathcal{S}_{i1}$ to be a strong first-order bisimulation. The proof for associativity shows $\mathcal{S}_a$ to be a strong first-order bisimulation up to $\sim^1$, by using the fact that $\sim^1$ is preserved by serial composition of stream processors. ∎

The dual of the restricted identity rule where **NullSP** is serially composed with a stream processor that can perform some output is not, however, equivalent to **NullSP**. This is because the output is buffered in the intermediate stream allowing the rightmost stream processor to continue and perhaps perform some input that is observable. Serial composition does not distribute over parallel composition:

$$(E >*< F) >==< G \not\approx^h (E >==< G) >*< (F >==< G) \quad (\mathit{Dist})$$

An example illustrating why this is the case is when

$$\begin{array}{rcl} E & = & \mathbf{GetSP}\ (\lambda x.\mathbf{if}\ x\ \mathbf{then\ PutSP\ true\ NullSP\ else\ NullSP}) \\ F & = & \mathbf{GetSP}\ (\lambda x.\mathbf{if}\ x\ \mathbf{then\ NullSP\ else\ PutSP\ false\ NullSP}) \\ G & = & (\mathbf{PutSP\ true\ NullSP}) >*< (\mathbf{PutSP\ false\ NullSP}) \end{array}$$

The left hand side of (Dist) can only make a single output transition whilst the right hand side has the possibility of making two or no output transitions as well.

We give only one axiom for the feedback construct for stream processors, establishing **NullSP** as an identity for it. There are no distribution rules for feedback as distributing over parallel composition would mean that the two branches of the parallel composition could not communicate with one another via the feedback loop. A similar problem arises with distribution over serial composition.

Theorem 9.4 (Feedback axioms)

$$\textbf{LoopSP NullSP} \sim^1 \textbf{NullSP} \quad (\textit{Identity})$$

Proof. *Neither term has any transitions and so the result follows trivially.* ∎

Finally, we consider the dynamic stream processor construct, **DynSP**, which does not distribute over parallel composition of stream processors. This is because when a new stream processor is sent to replace the old one then two copies of this stream processor would be created, one on each of the branches of the parallel composition. Distribution over serial composition fails for a similar reason, in that only one of the stream processors composed in series can be replaced dynamically. It also does not distribute over the feed construct, and there is no obvious identity for it.

Intuitively, it seems reasonable to expect that a serial composition of a stream processor, A, with the identity stream processor is just equivalent to A alone. However, this is not the case as we have that:

$$\begin{aligned} E >==< idSP &\not\sim^h E \\ idSP >==< F &\not\sim^h F \end{aligned}$$

Considering the first of these rules then if we take E to be **NullSP** then the left hand side of the rule can perform an input transition that the right hand side cannot match. The problem is that we can observe actions of the right hand branch of the parallel composition independently of the left hand branch.

If we take a demand driven approach to serial composition, where the right hand branch is only reduced to satisfy the demand for input of the left hand branch then the rules become valid. This constraint also eliminates the need for the feed construct in the transition rules for serial composition. In the case of a serial composition, any values communicated from the rightmost stream processor to

the leftmost must be processed immediately instead of being buffered in case they lead to the production of output. A similar approach can be taken with parallel composition, only allowing input when both branches are ready to perform input hence removing the need for any buffering of input. However, the feed construct is still required for feedback loops to model the buffer corresponding to the feedback loop itself. This approach is the one taken in the original Fudgets implementation.

One possibility for solving this problem such that the identity rules for serial composition hold is to add the following inference rule to the operational semantics:

$$(Inp_2)\quad \frac{, \vdash F : \tau}{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E \ll F}$$

This rule allows any term to perform an input transition, resulting in the original term with the input fed into it. For the first identity rule, if $E = \mathbf{NullSP}$ then the right hand side can now make a matching input transition as required.

9.2 Correctness of implementations

The operational semantics is also useful for checking the correctness of Core Fudget implementations. In order to check the correctness of an implementation we need to consider the implementation as a labelled transition system itself. If we can show this labelled transition system to *simulate* the labelled transition system of the operational semantics then we consider the implementation to be correct. Simulation in this case captures two properties:

- *Linear Time.* The operational semantics is nondeterministic but most implementations will be deterministic. Our definition of correctness does not aim to prescribe a particular scheduling strategy, and we consider the operational semantics as a loose definition of the required behaviour of an implementation. Most practical implementations will be incorrect if correctness is based upon branching time, and so our definition of correctness must respect linear time. This rules out using bisimulation as the basis of correctness, and instead leads naturally to considering simulation as a potential formalisation of correctness.

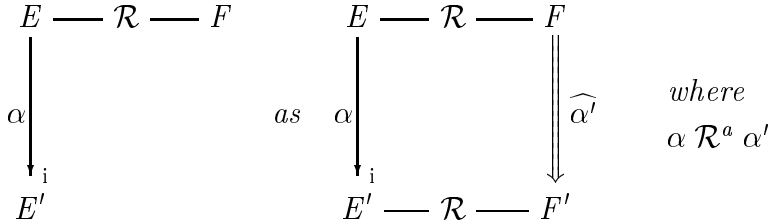
- *Termination.* Although we do not consider executions that do not exhibit all the concurrent computation captured by the operational semantics to be incorrect, we do insist on *maximal* executions. If a program terminates under an implementation then it must also do so under the operational semantics for the implementation to be correct. This implies that a reasonable definition of correctness must be sensitive to termination. We require this property as it makes trivial implementations incorrect.

These properties suggest that we base a formal definition of correctness on termination-preserving simulation relations between transition systems. However, we must also consider that an implementation may have been arrived at by a process of transition refinement from the operational semantics. As such the implementation may go through several transitions to accomplish what the operational semantics can do in a single transition. This motivates a definition of correctness using weak simulations, where we require that a transition of the implementation must be matched by a weak transition of the operational semantics. Firstly, we adapt the concept of weak higher-order simulation to work with the two transition systems, the one for the operational semantics, $\xrightarrow{\alpha}$, and the one corresponding to the implementation $\xrightarrow{\alpha}_i$. We also include a termination property to ensure that only maximal executions are considered:

Definition 9.1 (Correctness simulation) *$\mathcal{R}$ is a correctness simulation if it is a closed type-indexed family of relations that are structure-preserving and $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *if $\emptyset \vdash E : \tau \xrightarrow{\alpha}_i E'$ then $\exists F', \alpha'.\emptyset \vdash F : \tau \stackrel{\widehat{\alpha'}}{\Longrightarrow} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$ and $\emptyset \vdash \alpha\mathcal{R}^a\alpha' : \tau'$;*
- *if $\emptyset \vdash E : \tau \not\rightarrow_i$ then $\exists F'.\emptyset \vdash F : \tau \Longrightarrow F \not\rightarrow$.*

Correctness similarity $\lesssim$ is defined to be the greatest such relation. Intuitively, we can think of the first property as requiring that we can complete all diagrams of the following form (for simplicity types have been omitted):



Definition 9.2 (Correctness) *An implementation of Core Fudgets, characterised by a transition relation* $\xrightarrow{\alpha}_i$*, is correct with respect to the operational semantics if for all Core Fudget terms* $\emptyset \vdash E : \tau$ *then* $\emptyset \vdash E \lesssim E : \tau$.

We illustrate this definition by considering the stream processor subset of the original Fudgets implementation, showing that it is not correct. Firstly, we review the Haskell source code for the relevant parts of this implementation and develop a corresponding transition relation. Secondly, we examine the problems with this implementation that make it incorrect with respect to our operational semantics. Finally, we propose some alterations to the implementation that solve these problems, and show this altered implementation to be correct with respect to our operational semantics.

The original implementation represents stream processors by elements of the Haskell datatype shown in Figure 9.1. A stream processor that has terminated is represented by *NullSP*, whilst *PutSP* corresponds to a stream processor that can perform an output and continue as some other stream processor. The *GetSP* constructor corresponds to a stream processor that can read some input.

```
data SP i o = NullSP
            | PutSP o (SP i o)
            | GetSP (i → SP i o).
```

Figure 9.1: A representation of stream processors

We define the transition relation $\xrightarrow{\alpha}_i$ corresponding to the implementation in a similar manner to that used for the operational semantics in Chapter 8. The transitions are labelled with the same observations corresponding to input actions, output actions and internal administrative actions. The inference rules defining the transition relation are derived from the actual Haskell source code.

Firstly, we have transitions corresponding to the direct observations of elements of the above datatype. These transitions are shown in Figure 9.2. We note that there is no transition rule for **NullSP** and any terms equivalent to **NullSP** are reduced by the implementation to the term **NullSP** itself.

$$(Inp_1^i)\quad \frac{,\ \vdash ?F : \tau}{,\ \vdash \mathbf{GetSP}\ E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F}_i E\ F}$$

$$(Out^i)\quad ,\ \vdash \mathbf{PutSP}\ E\ F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!E}_i F$$

Figure 9.2: Atomic transitions

Next, we consider serial composition which uses a demand driven approach, only reducing the right hand side of the serial composition whenever the left hand side requires some input to continue. The Haskell definition in Figure 9.3 implements serial composition.

```
(>==<)                          ::  SP m o -> SP i m -> SP i o
NullSP      >==< f              =   NullSP
PutSP x k   >==< f              =   PutSP x (k >==< f)
GetSP f     >==< NullSP         =   NullSP
GetSP f     >==< PutSP x k      =   f x >==< k
GetSP f     >==< GetSP g        =   GetSP (\x -> GetSP f >==< g x)
```

Figure 9.3: The implementation of serial composition

The corresponding transition rules for serial composition are derived from the Haskell definition. We must be careful to maintain the semantics of Haskell pattern matching in these rules. This usually requires some extra transition rules to reduce arguments to a normal form of either **NullSP**, **GetSP**, or **PutSP**. The rules for serial composition are shown in Figure 9.4, with (Ser_6^i) and (Ser_7^i) reducing the left hand and right hand side of a serial composition to their normal forms.

As an example of the derivation of a transition rule from the Haskell source code, we consider the second line of the Haskell definition of serial composition. The corresponding transition rule, (Ser_2^i), has a hypothesis to ensure that the left hand side of the serial composition can perform an output action. The conclusion of the transition rule states that if this hypothesis is valid then the serial composition can itself perform the same output action. This derivation process captures the determinancy of the Haskell definitions in the corresponding transition rules.

$$(Ser_1^i) \qquad , \vdash \mathbf{NullSP} >==< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i \mathbf{NullSP}$$

$$(Ser_2^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{!G}_i E'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{!G}_i E' >==< F}$$

$$(Ser_3^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{?G}_i E'}{, \vdash E >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu}_i \mathbf{NullSP}}$$

$$(Ser_4^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{?G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G}_i F'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu}_i E' >==< F'}$$

$$(Ser_5^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{?G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?H}_i F'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{?H}_i E >==< F'}$$

$$(Ser_6^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{\nu}_i E'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu}_i E' >==< F}$$

$$(Ser_7^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{?G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i F'}{, \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu}_i E >==< F'}$$

Figure 9.4: Transition rules for serial composition

The implementation of parallel composition is shown in Figure 9.5. We note that the stream processor on the left hand side of the parallel composition is examined first, and thus if this stream processor continually produces output then the stream processor on the right hand side will never have a chance to execute. This implementation is unfair due to the elimination of nondeterminism by simply choosing to always bias towards one of the branches of a parallel composition.

$$
\begin{array}{llcl}
(>\!*\!<) & & :: & SP\ i\ o \rightarrow SP\ i\ o \rightarrow SP\ i\ o \\
NullSP & >\!*\!<\ f & = & f \\
e & >\!*\!<\ NullSP & = & e \\
PutSP\ x\ k & >\!*\!<\ f & = & PutSP\ x\ (k >\!*\!<\ f) \\
e & >\!*\!<\ PutSP\ x\ k & = & PutSP\ x\ (e >\!*\!<\ k) \\
GetSP\ f & >\!*\!<\ GetSP\ g & = & GetSP\ (\backslash x \rightarrow f\ x >\!*\!<\ g\ x)
\end{array}
$$

Figure 9.5: Parallel composition

The transition rules corresponding to this implementation of parallel composition are listed in Figure 9.6. The order of the lines in the Haskell definition is important. For example, in the second line e cannot be **NullSP**. These constraints are captured in the transition rules, with (Par_4^i), (Par_5^i) and (Par_6^i) reducing the left and right sides of a parallel composition to their normal forms in the same manner as the Haskell definition.

The feed construct is implemented by the source code shown in Figure 9.7. The first line of the definition states that feeding a term into **NullSP** is equivalent to **NullSP**. Feeding a term into a **PutSP** construct involves moving the feed into the continuation of the **PutSP** construct. Finally, feeding a term into a **GetSP** construct results in applying the function body of the **GetSP** construct to the term being fed. The corresponding transition rules for the feed construct are listed in Figure 9.8. In this case there is one transition rule for each line of the Haskell definition, and also an extra transition rule that reduces the stream processor argument to normal form.

The definition of the feedback construct given by Carlsson and Hallgren [CH98] includes the use of auxiliary functions modelling first-in first-out queues. These can be represented by the use of the feed construct directly as shown in Figure 9.9, with the corresponding transition rules in Figure 9.10.

$$(Par_1^i) \qquad , \vdash \mathbf{NullSP} >\!*\!< E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E$$

$$(Par_2^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha}_i E'}{, \vdash E >\!*\!< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha}_i E'} \quad \alpha \neq \nu$$

$$(Par_3^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha}_i F'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G}_i E' >\!*\!< F} \quad \alpha \neq \nu$$

$$(Par_4^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E' >\!*\!< F}$$

$$(Par_5^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i F'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E >\!*\!< F'}$$

$$(Par_6^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i F'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E >\!*\!< F'}$$

$$(Par_7^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!H}_i F'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!H}_i E >\!*\!< F'}$$

$$(Par_8^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G}_i E' \quad , \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G}_i F'}{, \vdash E >\!*\!< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G}_i E' >\!*\!< F'}$$

Figure 9.6: Transition rules for parallel composition

$$
\begin{array}{lcl}
\mathit{feedSP} & :: & i \to \mathit{SP}\ i\ o \to \mathit{SP}\ i\ o \\
\mathit{feedSP}\ x\ \mathit{NullSP} & = & \mathit{NullSP} \\
\mathit{feedSP}\ x\ (\mathit{PutSP}\ y\ k) & = & \mathit{PutSP}\ y\ (\mathit{feedSP}\ x\ k) \\
\mathit{feedSP}\ x\ (\mathit{GetSP}\ f) & = & f\ x
\end{array}
$$

Figure 9.7: The implementation of feed

$$
\begin{array}{ll}
(\mathit{Feed}_1^i) & , \vdash \mathbf{NullSP} \ll E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i \mathbf{NullSP} \\
(\mathit{Feed}_2^i) & \dfrac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G}_i E'}{, \vdash E \ll F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G}_i E' \ll F} \\
(\mathit{Feed}_3^i) & \dfrac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F}_i E'}{, \vdash E \ll F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E'} \\
(\mathit{Feed}_4^i) & \dfrac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E'}{, \vdash E \ll F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E' \ll F}
\end{array}
$$

Figure 9.8: Transition rules for feed

The dynamic stream processor construct is not implemented in the original Fudgets implementation. Instead, a similar construct is available at the Fudget level. However, it is straightforward to write a corresponding definition for stream processors, as shown in Figure 9.11. This implementation will produce output in preference to reading any input, including input indicating that the current stream processor should be replaced. The corresponding transition rules are shown in Figure 9.12.

We also need to consider transition rules for the basic computational mechanisms of the language, in this case we have rules for the call-by-name reduction mechanism of Haskell. Similarly, transition rules are required for literals and their corresponding constructs such as addition, and case analysis of binary sums. These transitions rules will be identical to those specified in the operational semantics in Chapter 8, and so we do not present them again.

```
loopSP                :: SP io io → SP io io
loopSP NullSP         =  NullSP
loopSP (PutSP x k)    =  PutSP x (loopSP (feedSP x k))
loopSP (GetSP f)      =  GetSP (\x → loopSP (f x))
```

Figure 9.9: The implementation of feedback

$$(Loop_1^i) \quad , \vdash \mathbf{LoopSP\ NullSP} : \mathbf{SP}\ \tau\ \tau \xrightarrow{\nu}_i \mathbf{NullSP}$$

$$(Loop_2^i) \quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau \xrightarrow{!G}_i E'}{, \vdash \mathbf{LoopSP}\ E : \mathbf{SP}\ \tau\ \tau \xrightarrow{!G}_i \mathbf{LoopSP}\ (E' \ll G)}$$

$$(Loop_3^i) \quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau \xrightarrow{?G}_i E'}{, \vdash \mathbf{LoopSP}\ E : \mathbf{SP}\ \tau\ \tau \xrightarrow{?G}_i \mathbf{LoopSP}\ E'}$$

$$(Loop_4^i) \quad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau \xrightarrow{\nu}_i E'}{, \vdash \mathbf{LoopSP}\ E : \mathbf{SP}\ \tau\ \tau \xrightarrow{\nu}_i \mathbf{LoopSP}\ E'}$$

Figure 9.10: Transition rules for feedback

```
dynSP                 :: SP (Either (SP i o) i) o
                            → SP (Either (SP i o) i) o
dynSP NullSP          =  GetSP (\x → case x of
                                       Left f    →  dynSP f
                                       Right _   →  dynSP NullSP)
dynSP (PutSP x k)     =  PutSP x (dynSP k)
dynSP (GetSP f)       =  GetSP (\x → case x of
                                       Left g    →  dynSP g
                                       Right x   →  dynSP (f x))
```

Figure 9.11: The implementation of dynamic stream processors

$$(Dyn_1^i) \qquad , \vdash \mathbf{DynSP\ NullSP} : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\mathbf{Right}\ E}_i \mathbf{DynSP\ NullSP}$$

$$(Dyn_2^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!F}_i E'}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{!F}_i \mathbf{DynSP}\ E'}$$

$$(Dyn_3^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F}_i E'}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\mathbf{Left}\ F}_i \mathbf{DynSP}\ F}$$

$$(Dyn_4^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F}_i E'}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\mathbf{Right}\ F}_i \mathbf{DynSP}\ E'}$$

$$(Dyn_5^i) \qquad \frac{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i E'}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{\nu}_i \mathbf{DynSP}\ E'}$$

$$(Dyn_6^i) \qquad \frac{\begin{array}{c}, \vdash F : \mathbf{SP}\ \tau\ \tau' + \tau \xrightarrow{\nu}_i F' \\ , \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?F'}_i E'\end{array}}{, \vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?F}_i E'}$$

Figure 9.12: Transition rules for dynamic stream processors

We now proceed to consider if this implementation is correct with respect to Definition 9.2. In fact we find that it is not correct due to two separate problems:

- The first problem is evident if we consider a serial composition of **NullSP** and some other stream processor. If we let the other stream processor be **GetSP** E then Definition 9.2 implies that the following diagram is valid:

$$\begin{array}{ccc} \emptyset \vdash \mathbf{NullSP} >==< \mathbf{GetSP}\ E & \lesssim & \mathbf{NullSP} >==< \mathbf{GetSP}\ E : \mathbf{SP}\ \tau\ \tau' \\ \Big\downarrow{\scriptstyle \nu}_{\mathrm{i}} & & \Big\Downarrow{\scriptstyle \widehat{\nu}} \\ \emptyset \vdash \mathbf{NullSP} & \lesssim & \mathbf{NullSP} >==< \mathbf{GetSP}\ E : \mathbf{SP}\ \tau\ \tau' \end{array}$$

  Although we have a matching weak transition, this also implies that we have $\emptyset \vdash \mathbf{NullSP} \lesssim \mathbf{NullSP} >==< \mathbf{GetSP}\ E : \mathbf{SP}\ \tau\ \tau'$. This is not the case because $\mathbf{NullSP} \not\rightarrow_i$ yet the other residual term has the transition $\emptyset \vdash \mathbf{NullSP} >==< \mathbf{GetSP}\ E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} \mathbf{NullSP} >==< E\ F$, and thus the termination-preserving property of $\lesssim$ is not met. A similar problem occurs if the right hand side of the serial composition is a stream processor that can perform some output.

- The second problem is with the dynamic stream processor construct, **DynSP**. Considering the term **DynSP NullSP** then Definition 9.2 requires that the following diagram is valid for some term F and action α:

$$\begin{array}{ccc} \emptyset \vdash \mathbf{DynSP}\ \mathbf{NullSP} & \lesssim & \mathbf{DynSP}\ \mathbf{NullSP} : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \\ \Big\downarrow{\scriptstyle ?\mathbf{Right}\ E}_{\mathrm{i}} & & \Big\Downarrow{\scriptstyle \widehat{\alpha}} \\ \emptyset \vdash \mathbf{DynSP}\ \mathbf{NullSP} & \lesssim & F : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \end{array}$$

  However, the operational semantics from Chapter 8 does not associate a transition with the term **DynSP NullSP**, and thus the only choice for F is **DynSP NullSP** yielding the reflexive transition where $\alpha = \nu$. Unfortunately, this does not satisfy the definition of implementation correctness because the actions ?**Right** E and ν do not match.

The first problem can be corrected by modifying the definition of serial composition of stream processors as shown in Figure 9.13. This involves the addition

of the first two lines which now allow input and output on the right-hand side of a serial composition when the left-hand side stream processor has terminated. The transition relation, $\xrightarrow{\alpha}_i$, is modified by replacing the (Ser_1^i) rule with the rules in Figure 9.14.

$$
\begin{array}{lll}
(>\!\!=\!\!=\!<) & :: & SP\ m\ o \to SP\ i\ m \to SP\ i\ o \\
NullSP\ >\!\!=\!\!=\!<\ GetSP\ g & = & GetSP\ (\backslash x \to NullSP >\!\!=\!\!=\!< g\ x) \\
NullSP\ >\!\!=\!\!=\!<\ PutSP\ x\ k & = & NullSP >\!\!=\!\!=\!<\ k \\
NullSP\ >\!\!=\!\!=\!<\ f & = & NullSP \\
PutSP\ x\ k >\!\!=\!\!=\!<\ f & = & PutSP\ x\ (k >\!\!=\!\!=\!< f) \\
GetSP\ f\ >\!\!=\!\!=\!<\ NullSP & = & NullSP \\
GetSP\ f\ >\!\!=\!\!=\!<\ PutSP\ x\ k & = & f\ x >\!\!=\!\!=\!< k \\
GetSP\ f\ >\!\!=\!\!=\!<\ GetSP\ g & = & GetSP\ (\backslash x \to GetSP\ f >\!\!=\!\!=\!< g\ x)
\end{array}
$$

Figure 9.13: The modified implementation of serial composition

$$
(Ser_1^i) \quad , \vdash \mathbf{NullSP} >\!\!=\!\!=\!< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i \mathbf{NullSP}
$$

$$
(Ser_8^i) \quad \frac{, \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha}_i F'}{, \vdash \mathbf{NullSP} >\!\!=\!\!=\!< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\alpha}_i \mathbf{NullSP} >\!\!=\!\!=\!< F'} \quad \alpha \neq !G
$$

$$
(Ser_9^i) \quad \frac{, \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G}_i F'}{, \vdash \mathbf{NullSP} >\!\!=\!\!=\!< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu}_i \mathbf{NullSP} >\!\!=\!\!=\!< F'}
$$

Figure 9.14: New serial composition transition rules

The second problem can be solved by altering the implementation such that the term **DynSP NullSP** has no transition. This corresponds to eliminating the third line of the Haskell definition for *dynSP*, and its associated transition rule (Dyn_1^i). This change can be justified because if one wants the behaviour that the (Dyn_1^i) transition rule yields then the following Haskell code can be used instead of **DynSP NullSP**:

$$
\begin{array}{lll}
dynNullSP & = & dynSP\ ignoreSP \\
 & & \textbf{where}\ ignoreSP\ =\ GetSP\ (\backslash\_ \to ignoreSP)
\end{array}
$$

Before showing this altered implementation to be correct according to Definition 9.2, we first prove an auxiliary lemma regarding input transitions:

Lemma 9.1 *If* , $\vdash E : \tau \xrightarrow{?F}_i E'$ *then the only possible transition under the operational semantics is by an input action, or that* , $\vdash E : \tau \not\xrightarrow{\nu}$ *and* , $\vdash E : \tau \not\xrightarrow{\downarrow G}$.

Proof. *By induction on the derivation of the transition* , $\vdash E : \tau \xrightarrow{?F}_i E'$. *We examine two of the cases with the others following in a similar manner:*

Case (Inp_1^i). *We have the derivation:*

$$\frac{, \vdash ?F : \tau}{, \vdash \mathbf{GetSP}\ E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F}_i E_1\ F}$$

The term , $\vdash \mathbf{GetSP}\ E_1 : \mathbf{SP}\ \tau\ \tau'$ *can only make transitions labelled by input actions under the operational semantics.*

Case (Ser_8^i). *We have the derivation:*

$$\frac{, \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F}_i E_2}{, \vdash \mathbf{NullSP} >==< E_1 : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{?F}_i \mathbf{NullSP} >==< E_2}$$

By induction the only possible transition of the term , $\vdash E_1 : \mathbf{SP}\ \tau\ \tau'$ *under the operational semantics is via an input action. From the* (Ser_1) *transition rule then the term* , $\vdash \mathbf{NullSP} >==< E_1 : \mathbf{SP}\ \tau\ \tau''$ *can also only perform a transition via an input action.* ∎

The concept of simulation up to can be adapted to correctness simulations and is useful in simplifying the proof of correctness of an implementation. Note that $\mathcal{S} \approx^h$ denotes the relational composition of $\mathcal{S}$ and $\approx^h$:

Definition 9.3 (Correctness simulation up to $\approx^h$) $\mathcal{R}$ *is a correctness simulation up to* $\approx^h$ *if it is a closed type-indexed family of relations that are structure-preserving and* $\emptyset \vdash E\mathcal{R}F : \tau$ *implies:*

- *if* $\emptyset \vdash E : \tau \xrightarrow{\alpha}_i E'$ *then* $\exists F', \alpha'.\emptyset \vdash F : \tau \xRightarrow{\widehat{\alpha'}} F'$ *and* $\emptyset \vdash E'\mathcal{R} \approx^h F' : \tau$ *and* $\emptyset \vdash \alpha\mathcal{R}^a\alpha' : \tau'$;
- *if* $\emptyset \vdash E : \tau \not\rightarrow_i$ *then* $\exists F'.\emptyset \vdash F : \tau \Longrightarrow F' \not\rightarrow$.

Lemma 9.2 *If $\mathcal{S}$ is a correctness simulation up to $\approx^h$ then the relational composition $\mathcal{S} \approx^h$ is also a correctness simulation.*

Proof. *Let $\emptyset \vdash E \mathcal{S} \approx^h F : \tau$ and $\emptyset \vdash E : \tau \xrightarrow{\alpha}_i E'$. It is enough to show that we can complete the following diagram:*

$$\begin{array}{lll} \emptyset \vdash E & \mathcal{S} \approx^h & F : \tau \\ \alpha \downarrow \mathrm{i} & & \Downarrow \widehat{\alpha'} \quad \text{where } \alpha(\mathcal{S} \approx^h)^a \alpha' \\ \emptyset \vdash E' & \mathcal{S} \approx^h & F' : \tau \end{array}$$

To do this, we first note that for some F_1 then $\emptyset \vdash E \mathcal{S} F_1 : \tau$ and $\emptyset \vdash F_1 \approx^h F : \tau$. Thus we can complete the following two diagrams in sequence, from left to right, knowing that $\mathcal{S}$ is a correctness simulation up to $\approx^h$:

$$\begin{array}{lll} \emptyset \vdash E & \mathcal{S} & F_1 : \tau \\ \alpha \downarrow \mathrm{i} & & \Downarrow \widehat{\alpha'} \quad \text{where } \alpha \mathcal{S}^a \alpha' \\ \emptyset \vdash E' & \mathcal{S} \approx^h & F_1' \end{array} \qquad \begin{array}{lll} \emptyset \vdash F_1 & \approx^h & F : \tau \\ \Downarrow \widehat{\alpha'} & & \Downarrow \widehat{\alpha''} \quad \text{where } \alpha' \approx^{h\,a} \alpha'' \\ \emptyset \vdash F_1' & \approx^h & F' : \tau \end{array}$$

Composing these, using the transitivity of $\approx^h$, we easily obtain the required diagram. A similar argument also holds for the termination property, and it is straightforward to show the relational composition of two structure-preserving type-indexed relations to also be structure-preserving. ∎

Lemma 9.3 *If $\mathcal{S}$ is a correctness simulation up to $\approx^h$ then $\mathcal{S} \subseteq \lesssim$.*

Proof. *Since by lemma 9.2, $\mathcal{S} \approx^h$ is a correctness simulation then $\mathcal{S} \approx^h \subseteq \lesssim$. But since the identity relation $Id = \{(E, E) \mid , \vdash E \approx^h E : \tau\}$ is contained in weak higher-order bisimulation, $Id \subseteq \approx^h$, then we have that $\mathcal{S} \subseteq \mathcal{S} \approx^h$ and can conclude that $\mathcal{S} \subseteq \lesssim$ as required.* ∎

This lemma is important, because to show $\emptyset \vdash E \lesssim F : \tau$ we only need to find a correctness simulation up to $\approx^h$ containing the pair (E, F). Using this technique we can now proceed to show the altered implementation to be correct according to Definition 9.2.

Theorem 9.5 (Correctness) *The implementation of Core Fudgets characterised by the transition relation $\xrightarrow{\alpha}_i$ is correct with respect to the operational semantics.*

Proof. *Let the relation $\mathcal{S}$ be defined as:*

$$\mathcal{S} = \{(E, E) \mid \emptyset \vdash E : \tau\}$$

We proceed by showing this relation to be a correctness simulation up to $\approx^h$, and thus $\emptyset \vdash E \lesssim E : \tau$ as required. Firstly, it is straightforward to show this relation to be structure-preserving. Next we show it satisfies the transfer properties for a correctness simulation up to $\approx^h$ by proceeding by induction on the derivation of the transition $\emptyset \vdash E : \tau \xrightarrow{\alpha}_i E'$. We we illustrate the proof by considering a number of the cases:

Case (Inp_1^i). *We have the derivation:*

$$\frac{\emptyset \vdash ?F : \tau}{\emptyset \vdash \mathbf{GetSP}\ E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F}_i E_1\ F}$$

By using the (Inp_1) transition rule we can derive a matching transition, $\emptyset \vdash \mathbf{GetSP}\ E_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?F}}{\Longrightarrow} E_1\ F$ and $\emptyset \vdash E_1\ F \mathcal{S} \approx^h E_1\ F : \mathbf{SP}\ \tau\ \tau'$ as required.

Case (Out^i). *We have the derivation:*

$$\emptyset \vdash \mathbf{PutSP}\ E_1\ E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!E_1}_i E_2$$

By using the (Out) transition rule we can derive a matching transition, $\emptyset \vdash \mathbf{PutSP}\ E_1\ E_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!E_1}}{\Longrightarrow} E_2$ and $\emptyset \vdash E_2 \mathcal{S} \approx^h E_2 : \mathbf{SP}\ \tau\ \tau'$ as required.

Case (Ser_1^i). *We have the derivation:*

$$\emptyset \vdash \mathbf{NullSP} >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu}_i \mathbf{NullSP}$$

The only possible matching transition is due to the reflexivity of $\Longrightarrow$ giving $\emptyset \vdash \mathbf{NullSP} >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{\nu}}{\Longrightarrow} \mathbf{NullSP} >==< \mathbf{NullSP}$. Since both **NullSP** *and* **NullSP** $>==<$ **NullSP** *do not have any transitions then*

$\emptyset \vdash \mathbf{NullSP} \approx^h \mathbf{NullSP} >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'$ *and thus we have* $\emptyset \vdash \mathbf{NullSP}\mathcal{S} \approx^h \mathbf{NullSP} >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'$ *as required.*

Case (Ser_2^i). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{!G}_i E_1'}{\emptyset \vdash E_1 >==< E_2 : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{!G}_i E_1' >==< E_2}$$

By induction there is some E_1'' *such that* $\emptyset \vdash E_1 : \mathbf{SP}\ \tau'\ \tau'' \stackrel{\widehat{!G'}}{\Longrightarrow} E_1''$, *where* $\emptyset \vdash E_1'\mathcal{S} \approx^h E_1'' : \mathbf{SP}\ \tau'\ \tau''$, *and* $\emptyset \vdash G(\mathcal{S} \approx^h)^a\, G' : \tau''$. *Using the* (Ser_2) *transition rule we derive that* $\emptyset \vdash E_1 >==< E_2 : \mathbf{SP}\ \tau\ \tau'' \stackrel{\widehat{!G'}}{\Longrightarrow} E_1'' >==< E_2$. *Also since* $\emptyset \vdash E_1'\mathcal{S} \approx^h E_1'' : \mathbf{SP}\ \tau'\ \tau''$ *then* $\emptyset \vdash E_1' \approx^h E_1'' : \mathbf{SP}\ \tau'\ \tau''$. *Finally since* $\approx^h$ *is preserved by serial composition then we also have* $\emptyset \vdash E_1' >==< E_2\mathcal{S} \approx^h E_1'' >==< E_2 : \mathbf{SP}\ \tau\ \tau''$ *as required.*

Case (Ser_3^i). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{?G}_i E_1'}{\emptyset \vdash E_1 >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu}_i \mathbf{NullSP}}$$

We have a matching weak transition due to the reflexivity of $\Longrightarrow$ *giving* $\emptyset \vdash E_1 >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'' \Longrightarrow E_1 >==< \mathbf{NullSP}$. *Since we have* $\emptyset \vdash E_1 : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{?G}_i E_1'$ *then by Lemma 9.1 the only possible transitions of* E_1 *under the operational semantics are due to input actions. This allows us to show that* $\emptyset \vdash \mathbf{NullSP} \approx^h E_1 >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau''$. *From this we obtain* $\emptyset \vdash \mathbf{NullSP}\mathcal{S} \approx^h E_1 >==< \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau''$ *as required.*

The other cases follow in a similar manner.

Finally, we must consider the termination condition and show that for all Core Fudget terms if $\emptyset \vdash E : \tau \not\rightarrow_i$ *then there is some* E' *such that* $\emptyset \vdash E : \tau \Longrightarrow E' \not\rightarrow$. *For our particular implementation, we can show that if* $\emptyset \vdash E : \tau \not\rightarrow_i$ *then it must be that* $E = \mathbf{NullSP}$ *and thus the termination preserving property follows trivially. The proof proceeds by induction on the structure of* E*:*

Case $E = \mathbf{GetSP}\ E'$. *The* (Inp_1^i) *rule yields a transition for* E.

Case $E = \mathbf{PutSP}\ E'\ E''$. *The* (Out^i) *rule yields a transition for* E.

Case $E = \mathbf{NullSP}$. *There are no transitions for this term as required.*

Case $E = E' >==< E''$. *We consider the possibilities for* E' *and* E'' *in the following table, showing that* E *always has a transition. The entries in the table correspond to the transition rule that is applicable in each case:*

$E' >==< E''$	$E'' \not\rightarrow_i$	$E'' \xrightarrow{\nu}_i$	$E'' \xrightarrow{?F}_i$	$E'' \xrightarrow{!F}_i$
$E' \not\rightarrow_i$	(Ser_1^i)	(Ser_8^i)	(Ser_8^i)	(Ser_9^i)
$E' \xrightarrow{\nu}_i$	(Ser_6^i)	(Ser_6^i)	(Ser_6^i)	(Ser_6^i)
$E' \xrightarrow{?G}_i$	(Ser_3^i)	(Ser_7^i)	(Ser_5^i)	(Ser_4^i)
$E' \xrightarrow{!G}_i$	(Ser_2^i)	(Ser_2^i)	(Ser_2^i)	(Ser_2^i)

For the first row and column induction allows us to assume that $E' = \mathbf{NullSP}$ *and* $E'' = \mathbf{NullSP}$, *respectively. For the third row and fourth column it is sufficient to consider only the cases when* $G = F$.

Case $E = E' >*< E''$. *We consider the possibilities for* E' *and* E'' *in the following table, showing that* E *always has a transition. The entries in the table correspond to the transition rule that is applicable in each case:*

$E' >*< E''$	$E'' \not\rightarrow_i$	$E'' \xrightarrow{\nu}_i$	$E'' \xrightarrow{?F}_i$	$E'' \xrightarrow{!F}_i$
$E' \not\rightarrow_i$	(Par_1^i)	(Par_1^i)	(Par_1^i)	(Par_1^i)
$E' \xrightarrow{\nu}_i$	(Par_4^i)	(Par_4^i)	(Par_4^i)	(Par_4^i)
$E' \xrightarrow{?G}_i$	(Par_2^i)	(Par_5^i)	(Par_8^i)	(Par_7^i)
$E' \xrightarrow{!G}_i$	(Par_2^i)	(Par_6^i)	(Par_3^i)	(Par_3^i)

For the first row and column induction allows us to assume that $E' = \mathbf{NullSP}$ *and* $E'' = \mathbf{NullSP}$, *respectively. It is sufficient to consider only the case when* $G = F$ *for the entry in the third row and third column.*

Case $E = E' \ll E''$. *It is sufficient to consider the possible transitions of* E' *alone, and we detail the transitions for these cases in the following table:*

$E' \ll E''$	$E' \not\rightarrow_i$	$E' \xrightarrow{\nu}_i$	$E' \xrightarrow{?G}_i$	$E' \xrightarrow{!G}_i$
	$(Feed_1^i)$	$(Feed_4^i)$	$(Feed_3^i)$	$(Feed_2^i)$

For the first column, induction allows us to assume that $E' = \mathbf{NullSP}$.

The remaining cases all follow in a similar manner. ∎

Chapter 10

Summary and further work

In this dissertation we have shown how a theory of the Fudgets system can be developed using ideas from concurrency theory. Firstly, we identified a core language underlying the Fudgets system, and next we developed two separate formal semantics for this language. Finally, we illustrated the application of such theories by developing equational rules and checking the correctness of implementations.

The first formal semantics we considered assigns meanings to programs by a translation into the π-calculus. There are two main advantages with using the π-calculus as a metalanguage. Firstly, there has been a great deal of work on the theory of the π-calculus and we can leverage this work to indirectly give a theory to Fudget programs. Secondly, the translation of Fudget programs into the π-calculus can be used as a reference implementation of the Fudgets system. However, the π-calculus is quite a low-level language and this makes proofs about Fudget programs long and tedious. This leads to the intuition behind the semantics of various core language constructs being lost in the π-calculus translations.

The second formal semantics we developed associates meanings to programs by operational transitions. This approach is much more direct than the first, but required a new theory to be developed. Various equivalences based on bisimulation were examined and weak higher-order bisimulation was chosen as an appropriate equivalence. This was shown to be a congruence, which is an important property for the equivalence because it allows it to be used as the basis of an equational theory where a subterm can be replaced by an equivalent subterm.

Finally, we considered applications of the formal semantics, specifically the second operational semantics. A set of equational rules were developed that are useful

for reasoning about Fudget programs without recourse to the complex machinery of bisimulation. A notion of implementation correctness was defined with respect to the operational semantics, and the original implementation of the Fudgets system was shown to be correct modulo some minor modifications.

10.1 Further work

We conclude the dissertation by describing possible future work, outlining preliminary ideas in each case.

Semantics of feedback: In Section 7.4 we discussed two possibilities for the semantics of the merge operation on the input side of the feedback construct **LoopSP**. The denotational semantics of Chapter 6 gives this merge operation a nondeterministic semantics, whilst in the operational semantics it has a deterministic semantics. Ideally, we would have liked to give the merge operation a nondeterministic semantics in the operational semantics. However, it is not obvious how to modify the $(Loop_2)$ transition rule from Chapter 8 such that this is the case. This would be an interesting area for further work.

Equivalence of semantics: Chapter 6 describes a denotational semantics for the Core Fudgets language, whilst Chapter 8 presents an operational semantics. An obvious question is how these two semantics relate to one another. However, to answer this question the feed construct described in Chapter 8 must be given an encoding in terms of the π-calculus. Comparing the two semantics is really an issue of comparing two operational semantics, the first being that induced by the denotational semantics and the operational semantics of the π-calculus. The second is simply the direct operational semantics of Core Fudgets from Chapter 8.

Comparison to other systems: Now that we have developed some formal theories for the Fudgets system, another possible application would be to use them to compare the Fudgets system to other graphical development systems. In particular, since we have a π-calculus semantics for the Fudgets system then this may be useful in comparing the Fudgets system to the Haggis system [FJ95]. This is because the Haggis system is written in Concurrent Haskell, for which

a formal semantics has been developed based on the π-calculus thus inducing a formal semantics for the Haggis system itself. In particular, this comparison may lead to some formal quantification of the relative expressiveness of the two systems.

Deriving implementations: We show in Chapter 9 how the operational semantics for Core Fudgets can be used to verify the correctness of implementations of the language. Another application of the semantics, that is closely related to correctness, could be to derive a correct implementation. This could be achieved by using a concurrent variant of Haskell such as Concurrent Haskell. This would enable the operational semantics to be encoded almost directly into the functional language. However, if there are no concurrent constructs in the target language then this would be more difficult. The nondeterminism of constructs such as parallel composition of stream processors could be eliminated by arbitrarily biasing to either the left or right stream processor in the composition. A notion of fairness would ideally need to be introduced in order to check whether such implementations were reasonable.

Lack of identities for serial composition: The equational rules we developed in Chapter 9 did not include standard identity rules for serial composition. Intuitively, we would expect the identity stream processor to be the identity for serial composition, but we showed this not to be the case. One solution is to adopt a demand driven approach as done in the original implementation of the Fudgets system. An alternative approach may be to extend the operational semantics with the (Inp_2) inference rule as discussed in Section 9.1:

$$(Inp_2)\quad \frac{, \vdash F : \tau}{, \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E \ll F}$$

The identity stream processor turns out to be an identity for serial composition under this extended semantics. However, some of the technical results we have developed need to be verified for this extended system, although it is straightforward to verify that the main result – the proof of congruence for weak higher-order bisimulation – still holds with this new transition rule. An investigation into the properties of this extended system is also a useful direction for further work, and we would hope that all of the equational rules from Chapter 9 still hold.

Appendix A

Proof of theorem 8.1

In this appendix we show that higher-order weak bisimulation, $\approx$, is a congruence for the Core Fudgets language, and is thus preserved by all of the languages constructs. The proof proceeds by a typed reworking of Howe's method [How89].

A.1 Overview of the proof

The proof of congruence follows a technique proposed by Howe [How89] which was first used in the context of applicative simulation equivalences for the lazy λ-calculus. The core of this method involves the definition of a closure operator, $(\_)^\bullet$, which by definition is a congruence. Three steps are then required to obtain the desired congruence result:

- *Establishing standard properties.* A set of standard properties are shown to be valid for the closure operator $(\_)^\bullet$. These properties are used in the remaining two steps of the proof.
- *Proving a strengthened simulation property.* This is the main step of the proof, and involves showing the relation $\approx^\bullet$ to satisfy a strengthened form of the properties required for a relation to be a weak higher-order simulation.
- *Concluding the proof.* The relation $\approx^\bullet$ is shown to be a weak higher-order bisimulation using an observation of Howe's regarding the closure operator $(\_)^\bullet$. Finally, a set inclusion argument is used to show $\approx^\bullet$ to be equal to $\approx^\circ$, and hence $\approx^\circ$ is a congruence.

A.2 Compatible refinement

Howe's method for proving a bisimulation relation to be a congruence makes use of an alternative characterisation of congruence based upon the concept of *compatible refinement*. The compatible refinement of a relation $\mathcal{R}$ relates those expressions that have the same toplevel syntactic constructor and whose corresponding subterms are related by $\mathcal{R}$. Formally, we define compatible refinement as:

Definition A.1 (Compatible refinement) *For an open type-indexed relation R on Core Fudget terms, we define its compatible refinement, $\widehat{\mathcal{R}}$, as the least relation defined by the inference rules in Figures A.1 , A.2, and A.3.*

$$(\mathit{Comp\ Lit}) \qquad , \vdash E \,\widehat{\mathcal{R}}\, E : \tau \qquad \text{if } E \text{ is a literal}$$

$$(\mathit{Comp\ Var}) \qquad , \vdash x \,\widehat{\mathcal{R}}\, x : \tau$$

$$(\mathit{Comp\ Abs}) \qquad \frac{, , x : \tau \vdash E \,\mathcal{R}\, E' : \tau'}{, \vdash \lambda x.E \,\widehat{\mathcal{R}}\, \lambda x.E' : \tau \to \tau'}$$

$$(\mathit{Comp\ App}) \qquad \frac{, \vdash E \,\mathcal{R}\, E' : \tau \to \tau' \quad , \vdash F \,\mathcal{R}\, F' : \tau}{, \vdash E\ F \,\widehat{\mathcal{R}}\, E'\ F' : \tau'}$$

$$(\mathit{Comp\ Fix}) \qquad \frac{, , x : \tau \vdash E \,\mathcal{R}\, E' : \tau}{, \vdash \mathbf{Fix}\ x.E \,\widehat{\mathcal{R}}\, \mathbf{Fix}\ x.E' : \tau}$$

Figure A.1: Compatible refinement for literals and expressions

An alternative definition of congruence to that given in Definition 8.4 can now be formulated in terms of compatible refinement:

Definition A.2 (Congruence$_2$) *A relation $\mathcal{R}$ is a* precongruence *if and only if it contains its own compatible refinement, $\widehat{\mathcal{R}} \subseteq \mathcal{R}$. A* congruence *is an equivalence relation that is a precongruence.*

$$(\mathit{Comp\ Null}) \qquad , \vdash \mathbf{NullSP}\ \widehat{\mathcal{R}}\ \mathbf{NullSP} : \mathbf{SP}\ \tau\ \tau'$$

$$(\mathit{Comp\ Get}) \qquad \frac{, \vdash E\ \mathcal{R}\ E' : \tau \to \mathbf{SP}\ \tau\ \tau'}{, \vdash \mathbf{GetSP}\ E\ \widehat{\mathcal{R}}\ \mathbf{GetSP}\ E' : \mathbf{SP}\ \tau\ \tau''}$$

$$(\mathit{Comp\ Put}) \qquad \frac{, \vdash E\ \mathcal{R}\ E' : \tau' \quad , \vdash F\ \mathcal{R}\ F' : \mathbf{SP}\ \tau\ \tau'}{, \vdash \mathbf{PutSP}\ E\ F\ \widehat{\mathcal{R}}\ \mathbf{PutSP}\ E'\ F' : \mathbf{SP}\ \tau\ \tau'}$$

$$(\mathit{Comp\ Par}) \qquad \frac{, \vdash E\ \mathcal{R}\ E' : \mathbf{SP}\ \tau\ \tau' \quad , \vdash F\ \mathcal{R}\ F' : \mathbf{SP}\ \tau\ \tau'}{, \vdash E >\!*\!<\ F\ \widehat{\mathcal{R}}\ E' >\!*\!<\ F' : \mathbf{SP}\ \tau\ \tau'}$$

$$(\mathit{Comp\ Ser}) \qquad \frac{, \vdash E\ \mathcal{R}\ E' : \mathbf{SP}\ \tau'\ \tau'' \quad , \vdash F\ \mathcal{R}\ F' : \mathbf{SP}\ \tau\ \tau'}{, \vdash E >\!=\!=\!<\ F\ \widehat{\mathcal{R}}\ E' >\!=\!=\!<\ F' : \mathbf{SP}\ \tau\ \tau''}$$

$$(\mathit{Comp\ Loop}) \qquad \frac{, \vdash E\ \mathcal{R}\ E' : \mathbf{SP}\ \tau\ \tau}{, \vdash \mathbf{LoopSP}\ E\ \widehat{\mathcal{R}}\ \mathbf{LoopSP}\ E' : \mathbf{SP}\ \tau\ \tau}$$

$$(\mathit{Comp\ Dyn}) \qquad \frac{, \vdash E\ \mathcal{R}\ E' : \mathbf{SP}\ \tau\ \tau'}{, \vdash \mathbf{DynSP}\ E\ \widehat{\mathcal{R}}\ \mathbf{DynSP}\ E' : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'}$$

$$(\mathit{Comp\ Feed}) \qquad \frac{, \vdash E\ \mathcal{R}\ E' : \mathbf{SP}\ \tau\ \tau' \quad , \vdash F\ \mathcal{R}\ F' : \tau}{, \vdash E \ll F\ \widehat{\mathcal{R}}\ E' \ll F' : \mathbf{SP}\ \tau\ \tau'}$$

Figure A.2: Compatible refinement for stream processors

$$(\mathit{Comp\ LInj}) \quad \frac{\Gamma, \Gamma' \vdash E\ \mathcal{R}\ E' : \tau}{\Gamma, \Gamma' \vdash \mathbf{Left}\ E\ \widehat{\mathcal{R}}\ \mathbf{Left}\ E' : \tau + \tau'}$$

$$(\mathit{Comp\ RInj}) \quad \frac{\Gamma, \Gamma' \vdash F\ \mathcal{R}\ F' : \tau'}{\Gamma, \Gamma' \vdash \mathbf{Right}\ F\ \widehat{\mathcal{R}}\ \mathbf{Right}\ F' : \tau + \tau'}$$

$$(\mathit{Comp\ Case}) \quad \frac{\Gamma, \Gamma' \vdash E\ \mathcal{R}\ E' : \tau + \tau' \quad \Gamma, \Gamma' \vdash F\ \mathcal{R}\ F' : \tau \to \tau'' \quad \Gamma, \Gamma' \vdash G\ \mathcal{R}\ G' : \tau' \to \tau''}{\Gamma, \Gamma' \vdash \mathbf{Case}\ E \to F, G\ \widehat{\mathcal{R}}\ \mathbf{Case}\ E' \to F', G' : \tau''}$$

$$(\mathit{Comp\ If}) \quad \frac{\Gamma, \Gamma' \vdash E\ \mathcal{R}\ E' : \mathbf{bool} \quad \Gamma, \Gamma' \vdash F\ \mathcal{R}\ F' : \tau \quad \Gamma, \Gamma' \vdash G\ \mathcal{R}\ G' : \tau}{\Gamma, \Gamma' \vdash \mathbf{if}\ E\ \mathbf{then}\ F\ \mathbf{else}\ G\ \widehat{\mathcal{R}}\ \mathbf{if}\ E'\ \mathbf{then}\ F'\ \mathbf{else}\ G' : \tau}$$

$$(\mathit{Comp\ Add}) \quad \frac{\Gamma, \Gamma' \vdash E\ \mathcal{R}\ E' : \mathbf{num} \quad \Gamma, \Gamma' \vdash F\ \mathcal{R}\ F' : \mathbf{num}}{\Gamma, \Gamma' \vdash \mathbf{add}\ E\ F\ \widehat{\mathcal{R}}\ \mathbf{add}\ E'\ F' : \mathbf{num}}$$

$$(\mathit{Comp\ Equal}) \quad \frac{\Gamma, \Gamma' \vdash E\ \mathcal{R}\ E' : \mathbf{num} \quad \Gamma, \Gamma' \vdash F\ \mathcal{R}\ F' : \mathbf{num}}{\Gamma, \Gamma' \vdash \mathbf{equal}\ E\ F\ \widehat{\mathcal{R}}\ \mathbf{equal}\ E'\ F' : \mathbf{bool}}$$

Figure A.3: Compatible refinement for constant functions

Recalling the earlier definition of congruence in terms of contexts, we can show that this is equivalent to our formulation of congruence in terms of compatible refinement:

Lemma A.1 *The two characterisations of congruence from Definitions 8.4 and A.2 are equivalent.*

Proof. *It is straightforward to show that the inference rules characterising compatible refinement can be derived from* (*Cong* $\mathcal{R}$) *and by using the properties of the relation* $\mathcal{R}$*, such as reflexivity and transitivity. To show that* $\widehat{\mathcal{R}} \subseteq \mathcal{R}$ *implies* (*Cong* $\mathcal{R}$) *requires using induction on the derivation of* $\Gamma, \Gamma' \vdash \mathcal{C}[-_\tau] : \tau'$ *to show that if* Γ, Γ' *is the list of bound variables in the context* $\mathcal{C}$ *then* $\Gamma, \Gamma, \Gamma' \vdash E \mathcal{R} E' : \tau$ *implies that* $\Gamma, \Gamma' \vdash \mathcal{C}[E] \mathcal{R} \mathcal{C}[E'] : \tau'$*. We consider the case for the* **GetSP** *construct:*

Case $\Gamma, \Gamma' \vdash \mathbf{GetSP}\ \mathcal{C}' : \mathbf{SP}\ \tau\ \tau'$*. By induction, the result holds for* $\mathcal{C}'$ *and so we have the derivation* $\Gamma, \Gamma' \vdash \mathcal{C}'[E]\ \mathcal{R}\ \mathcal{C}'[E'] : \tau \to \mathbf{SP}\ \tau\ \tau'$*, and using* (*Comp Get*) *we obtain* $\Gamma, \Gamma' \vdash \mathbf{GetSP}\ \mathcal{C}'[E]\ \widehat{\mathcal{R}}\ \mathbf{GetSP}\ \mathcal{C}'[E'] : \mathbf{SP}\ \tau\ \tau'$*, as required.*

The other cases follow in a similar manner. ∎

A.3 Compatible closure

The core of Howe's method is the definition of a special relation, called the compatible closure, which by definition is a congruence. The main idea is then to show that the relation one wishes to prove to be a congruence is equal to the compatible closure of that relation.

Definition A.3 (Compatible closure) *The compatible closure of a type-indexed relation, $\mathcal{R}$, is denoted by $\mathcal{R}^\bullet$ and is defined by the following inference rule:*

$$(\textit{Cand Def}) \quad \frac{, \vdash E\ \widehat{\mathcal{R}^\bullet}\ E'' : \tau \qquad , \vdash E''\ \mathcal{R}^\circ\ E' : \tau}{, \vdash E\ \mathcal{R}^\bullet\ E' : \tau}$$

This definition is both circular and asymmetric, and it may appear that the circularity will lead to empty relations, but this is shown not to be the case in the following lemma. The asymmetry of the definition suggests that it may only be useful for showing non-symmetric relations to be congruences, but this problem can be overcome by using a recent observation of Howe's, and we will return to this point in a later section.

Lemma A.2 (Properties of compatible closure) *If $\mathcal{R}$ is a preorder then its compatible closure $\mathcal{R}^\bullet$ satisfies the inference rules listed in Figure A.4.*

$$(\textit{Cand Refl}) \qquad , \vdash E\ \mathcal{R}^\bullet\ E : \tau$$

$$(\textit{Cand Cong}) \qquad \frac{, \vdash E\ \widehat{\mathcal{R}^\bullet}\ E' : \tau}{, \vdash E\ \mathcal{R}^\bullet\ E' : \tau}$$

$$(\textit{Cand Right}) \qquad \frac{, \vdash E\ \mathcal{R}^\bullet\ E'' : \tau \qquad , \vdash E''\ \mathcal{R}^\circ\ E' : \tau}{, \vdash E\ \mathcal{R}^\bullet\ E' : \tau}$$

$$(\textit{Cand Sim}) \qquad \frac{, \vdash E\ \mathcal{R}^\circ\ E' : \tau}{, \vdash E\ \mathcal{R}^\bullet\ E' : \tau}$$

$$(\textit{Cand Subst}) \qquad \frac{, , x : \tau \vdash E\ \mathcal{R}^\bullet\ E' : \tau' \qquad , \vdash F\ \mathcal{R}^\bullet\ F' : \tau}{, \vdash E[F/x]\ \mathcal{R}^\bullet\ E'[F'/x] : \tau'}$$

Figure A.4: Properties of compatible closure

Proof. *We consider the proof of each inference rule separately:*

(*Cand Refl*). *By using the reflexivity of* $\mathcal{R}$, *we can use structural induction on* E *to show that* , $\vdash E\ \mathcal{R}^\bullet\ E : \tau$.

(*Cand Cong*). *By using* (*Cand Def*), *setting* $E'' = E'$, *and by the reflexivity of* $\mathcal{R}^\circ$ *the result follows.*

(*Cand Right*). *From* , $\vdash E\,\mathcal{R}^\bullet\,E'' : \tau$ *and by using the* (*Cand Def*) *rule there must be some* E''' *such that* , $\vdash E\,\widehat{\mathcal{R}^\bullet}\,E''' : \tau$ *and* , $\vdash E'''\,\mathcal{R}^\circ\,E'' : \tau$. *The transitivity of* $\mathcal{R}^\circ$ *allows us to deduce* , $\vdash E'''\ \mathcal{R}^\circ\ E' : \tau$, *and using the* (*Cand Def*) *rule yields* , $\vdash E\ \mathcal{R}^\bullet\ E' : \tau$ *as required.*

(*Cand Sim*). *Since* $\mathcal{R}^\bullet$ *is reflexive by* (*Cand Refl*) *then we have that* , $\vdash E\mathcal{R}^\bullet E : \tau$ *and combining this with the hypothesis of* (*Cand Sim*), , $\vdash E\,\mathcal{R}^\circ\,E' : \tau$, *using the* (*Cand Right*) *rule we obtain* , $\vdash E\ \mathcal{R}^\bullet\ E' : \tau$ *as required.*

(*Cand Subst*). *The proof proceeds by induction on the structure of the derivation of* , , $x : \tau \vdash E\ \mathcal{R}^\bullet\ E' : \tau'$. *By definition from the* (*Cand Def*) *rule there is some* E'' *such that* , , $x : \tau \vdash E\ \widehat{\mathcal{R}^\bullet}\ E'' : \tau'$ *and* , , $x : \tau \vdash E''\ \mathcal{R}^\circ\ E' : \tau'$. *From the definition of* $\mathcal{R}^\circ$ *on open terms we have* , $\vdash E''[F'/x]\mathcal{R}^\circ E'[F'/x] : \tau'$. *We proceed by structural analysis of* E *and show that in each case the judgement* , $\vdash E[F/x]\ \mathcal{R}^\bullet\ E''[F'/x] : \tau'$ *can be derived:*

Case $E = \mathbf{GetSP}\ G$. *From the definition of* $\widehat{\mathcal{R}^\bullet}$ *then* $E'' = \mathbf{GetSP}\ G'$, *where* , , $x : \tau \vdash G\,\mathcal{R}^\bullet\,G' : \tau'' \to \tau'$. *By induction we have the derivation* , $\vdash G[F/x]\ \mathcal{R}^\bullet\ G'[F'/x] : \tau'' \to \tau'$, *and so by using the* (*Comp Get*) *rule we obtain that* , $\vdash \mathbf{GetSP}\ (G[F/x])\ \widehat{\mathcal{R}^\bullet}\ \mathbf{GetSP}\ (G'[F'/x]) : \tau'$. *The rules of substitution along with the* (*Cand Cong*) *rule give us the required result,* , $\vdash (\mathbf{GetSP}\ G)[F/x]\ \mathcal{R}^\bullet\ (\mathbf{GetSP}\ G')[F'/x] : \tau'$.

The other cases follow in a similar manner.

■

The properties listed in Figure A.4 are standard, and will be required in the next section to show the equivalence of a relation and its compatible closure. The (*Cand Subst*) rule is especially important in the proof technique as it is necessary in handling the synchronisation cases involving substitution that cause problems for proof techniques that work for first-order calculi.

A.4 Congruence of $\approx$

We are now in a position to prove the open extension of higher-order weak bisimulation to be a congruence. If we consider the compatible closure of higher-order weak bisimulation $\approx^{h\bullet}$, then by definition this is a congruence as it contains its own compatible refinement. The result will follow if we can show that $\approx^{h\circ} \subseteq \approx^{h\bullet}$ and $\approx^{h\bullet} \subseteq \approx^{h\circ}$. The first of these equations follows directly from the (*Cand Sim*) rule, while the second requires further work.

We approach the problem of showing $\approx^{h\bullet} \subseteq \approx^{h\circ}$, by considering the restriction of $\approx^{h\bullet}$ to closed terms as captured by the following relation:

$$\mathcal{S} = \{(E, F) \mid \emptyset \vdash E \approx^{h\bullet} F : \tau\}$$

First, we show that this relation is structure-preserving:

Lemma A.3 *The relation $\mathcal{S}$ is structure-preserving.*

Proof. *There are four cases to consider:*

Case $\emptyset \vdash L \approx^{h\bullet} E : \tau$, *where* $\tau \in \{\mathbf{bool}, \mathbf{num}\}$. *From* (*Cand Def*) *there must be some* E' *such that* $\emptyset \vdash L \widehat{\approx^{h\bullet}} E' : \tau$ *where* $\emptyset \vdash E' \approx^{h\circ} E : \tau$. *However, because* L *is a literal then the only rule that can be used to infer the first of these judgements is the* (*Comp Lit*) *rule, and thus* $E' = L$. *Further, from the definition of open extension then we have* $\emptyset \vdash E' \approx^{h} E : \tau$ *and as* $\approx^{h}$ *is structure-preserving by definition then there must be some* M *such that* $\emptyset \vdash E : \tau \Longrightarrow M$ *and* $L = M$ *as required.*

Case $\emptyset \vdash \mathbf{Left}\ E \approx^{h\bullet} F : \tau + \tau'$. *From the* (*Cand Def*) *rule there is some* G *such that* $\emptyset \vdash \mathbf{Left}\ E \widehat{\approx^{h\bullet}} G : \tau + \tau'$ *and* $\emptyset \vdash G \approx^{h\circ} F : \tau + \tau'$. *By the definition of compatible refinement then there is some* G' *such that* $G = \mathbf{Left}\ G'$ *where* $\emptyset \vdash E \approx^{h\bullet} G' : \tau$. *Further, from the definition of open extension then we have that* $\emptyset \vdash G \approx^{h} F : \tau + \tau'$ *and because* $\approx^{h}$ *is structure-preserving then there is some* F' *such that* $\emptyset \vdash F : \tau + \tau' \Longrightarrow \mathbf{Left}\ F'$ *where* $\emptyset \vdash G' \approx^{h} F' : \tau$. *Finally, from the definition of open extension this is equivalent to* $\emptyset \vdash G' \approx^{h\circ} F' : \tau$ *and by using* (*Cand Right*) *we can thus show that* $\emptyset \vdash E \approx^{h\bullet} F' : \tau$ *as required.*

Case $\emptyset \vdash \mathbf{Right}\ E \approx^{h^\bullet} F : \tau + \tau'$. *From the* (*Cand Def*) *rule there is some* G *such that* $\emptyset \vdash \mathbf{Right}\ E \widehat{\approx^{h^\bullet}} G : \tau + \tau'$ *and* $\emptyset \vdash G \approx^{h^\circ} F : \tau + \tau'$. *By the definition of compatible refinement then there is some* G' *such that* $G = \mathbf{Right}\ G'$ *and* $\emptyset \vdash E \approx^{h^\bullet} G' : \tau'$. *Further, from the definition of open extension then we have* $\emptyset \vdash G \approx^{h} F : \tau + \tau'$ *and as* $\approx^h$ *is structure-preserving then there is some* F' *such that* $\emptyset \vdash F : \tau + \tau' \Longrightarrow \mathbf{Right}\ F'$ *where* $\emptyset \vdash G' \approx^h F' : \tau'$. *Finally, from the definition of open extension this is equivalent to* $\emptyset \vdash G' \approx^{h^\circ} F' : \tau'$ *and by using* (*Cand Right*) *we can thus show that* $\emptyset \vdash E \approx^{h^\bullet} F' : \tau'$ *as required.*

Case $\emptyset \vdash (\lambda x.E) \approx^{h^\bullet} F : \tau \to \tau'$. *From* (*Cand Def*) *there must be some* G *such that* $\emptyset \vdash (\lambda x.e) \widehat{\approx^{h^\bullet}} G : \tau \to \tau'$ *where* $\emptyset \vdash G \approx^{h^\circ} F : \tau \to \tau'$. *From the* (*Comp Abs*) *rule then there must be some* G' *such that* $G = (\lambda x.G')$ *where* $x : \tau \vdash E \approx^{h^\bullet} G' : \tau'$. *Using the* (*Cand Cong*) *rule we obtain the derivation* $\emptyset \vdash (\lambda x.E) \approx^{h^\bullet} (\lambda x.G') : \tau \to \tau'$, *and from the* (*Cand Refl*) *rule we have* $y : \tau \vdash y \approx^{h^\bullet} y : \tau$. *Thus, by using the* (*Comp App*) *and* (*Cand Cong*) *rules we can infer that* $y : \tau \vdash (\lambda x.E)\ y \approx^{h^\bullet} (\lambda x.G')\ y : \tau'$. *For all* $\emptyset \vdash H : \tau$ *using the* (*Cand Refl*) *rule we can show that* $\emptyset \vdash H \approx^{h^\bullet} H : \tau$ *and thus from* (*Cand Subst*) *we can derive* $\emptyset \vdash (\lambda x.E)\ H \approx^{h^\bullet} (\lambda x.G')\ H : \tau'$. *Further, from the definition of open extension then* $\emptyset \vdash G \approx^h F : \tau \to \tau'$ *and as* $\approx^h$ *is structure-preserving by definition then there must be some* F' *such that* $\emptyset \vdash F : \tau \to \tau' \Longrightarrow (\lambda x.F')$, *where for all terms* $\emptyset \vdash H' : \tau$ *then we have* $\emptyset \vdash (\lambda x.G')\ H' \approx^h (\lambda x.F')\ H' : \tau$. *The definition of open extension allows us to infer that* $\emptyset \vdash (\lambda x.G')\ H' \approx^{h^\circ} (\lambda x.F')\ H' : \tau$. *We can instantiate* H' *to be* H *and thus for all terms* $\emptyset \vdash H : \tau$ *then by using* (*Cand Right*) *we have that* $(\lambda x.E)\ H \approx^{h^\bullet} (\lambda x.F')\ H \vdash \tau :$ *as required.*

■

We prove some auxiliary lemmas regarding the dynamic stream processor construct that will be required later in the proof:

Lemma A.4 *If* , $\vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G'$ *and* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\,G'} E'$ *then* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?\,G}}{\Longrightarrow} E'$.

Proof. *By induction on the length of the transition* , $\vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G'$*:*

Case *Base Case. In this case* $G = G'$ *and so the result follows trivially.*

Case *Inductive Case. There is some* G'' *such that* , $\vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G'$ *can be broken into the two separate transitions* , $\vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G''$ *and* , $\vdash G'' : \mathbf{SP}\ \tau\ \tau' + \tau \xrightarrow{\nu} G'$. *From the second of these transitions along with* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?G'} E'$ *using the* (Dyn_2) *transition rule we can conclude that* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?G''} E'$. *Now by induction we can show that* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G}}{\Longrightarrow} E'$ *as required.*

∎

Lemma A.5 *If* , $\vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G'$ *and* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{?G'}{\Longrightarrow} E'$ *then* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G}}{\Longrightarrow} E'$.

Proof. *From* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{?G'}{\Longrightarrow} E'$ *then there must be some* E'' *and* E''' *such that* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \Longrightarrow E''$, , $\vdash E'' : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?G'} E'''$ *and* , $\vdash \mathbf{DynSP}\ E''' : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \Longrightarrow E'$. *The only possible* ν *transitions for terms of the form* **DynSP** E *are due to the* (Dyn_1) *transition rule, and thus* $E'' = \mathbf{DynSP}\ E''''$. *Since* , $\vdash \mathbf{DynSP}\ E'''' : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?G'} E'''$ *and* , $\vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G'$ *then by Lemma A.4 we have that* , $\vdash \mathbf{DynSP}\ E'''' : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G}}{\Longrightarrow} E'''$. *Finally using the* (Dyn_1) *transition rule we can deduce that* , $\vdash \mathbf{DynSP}\ E : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G}}{\Longrightarrow} E'$ *as required.* ∎

Next, we show the relation $\mathcal{S}$ to satisfy a strengthened set of the transfer properties required for a relation to be a weak higher-order simulation. The difference in these properties and those required by weak higher-order simulation is in the treatment of input actions: instead of requiring an input action to be matched precisely we require there to be matching transitions for all semantically equivalent inputs. This modification to the transfer properties is critical to the success of showing $\mathcal{S}$ to satisfy the transfer properties of weak higher-order simulation. The properties of weak higher-order simulation are just one instance of these more general properties.

Lemma A.6 *The relation $\mathcal{S}$ satisfies the following strengthened transfer properties of weak higher-order simulation where $\emptyset \vdash E \mathcal{S} F : \tau$ implies:*

- *if $\emptyset \vdash E : \tau \xrightarrow{?G} E'$ then $\forall G'.\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ implies $\exists F'.\emptyset \vdash F : \tau \stackrel{\widehat{?G'}}{\Longrightarrow} F'$ and $\emptyset \vdash E' \approx^{h^\bullet} F' : \tau$;*
- *if $\emptyset \vdash E : \tau \xrightarrow{\nu} E'$ then $\exists F'.\emptyset \vdash F : \tau \Longrightarrow F'$ and $\emptyset \vdash E' \approx^{h^\bullet} F' : \tau$;*
- *if $\emptyset \vdash E : \tau \xrightarrow{!G} E'$ then $\exists F', G'.\emptyset \vdash F : \tau \stackrel{\widehat{!G'}}{\Longrightarrow} F'$ and $\emptyset \vdash E' \approx^{h^\bullet} F' : \tau$ and $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$.*

Proof. *By induction on the structure of the derivation of $\emptyset \vdash E \xrightarrow{\alpha} E' : \tau$. Since $\emptyset \vdash E \approx^{h^\bullet} F : \tau$, then from the (Cand Def) rule we know that there is some H such that $\emptyset \vdash E \stackrel{\widehat{}}{\approx}^{h^\bullet} H : \tau$ and $\emptyset \vdash H \approx^{h^\circ} F : \tau$. It is sufficient to show that there are appropriate transitions of H because if $\emptyset \vdash H : \tau \stackrel{\widehat{\alpha}}{\Longrightarrow} H'$ then from the definition of $\approx^{h^\circ}$ there must also be a transition $\emptyset \vdash F : \tau \stackrel{\widehat{\alpha'}}{\Longrightarrow} F'$ where $\emptyset \vdash \alpha \approx^{h^\circ{}^a} \alpha' : \tau''$ and $\emptyset \vdash H' \approx^{h^\circ} F' : \tau$. By using (Cand Right) and (Cand Cong) we can use this to relate the residual term of the transition of E to F' as required:*

Case (Inp_1). *We have the derivation:*

$$\frac{\emptyset \vdash ?G : \tau}{\emptyset \vdash \mathbf{GetSP}\ E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E_1\ G}$$

From the (Comp Get) rule there is some H_1 such that $H = \mathbf{GetSP}\ H_1$, where $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \tau \to \mathbf{SP}\ \tau\ \tau'$. From the (Inp Act) typing rule for actions we can infer that for all G' such that $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ then $\emptyset \vdash ?G' : \tau$. Using this with the ($Inp_1$) transition rule we can infer that for each G' then we have a derivation of the form:

$$\frac{\emptyset \vdash ?G' : \tau}{\emptyset \vdash \mathbf{GetSP}\ H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} H_1\ G'}$$

Finally, because $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \tau \to \mathbf{SP}\ \tau\ \tau'$ and $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ then using (Comp App) and (Cand Cong) we obtain $\emptyset \vdash E_1\ G \approx^{h^\bullet} H_1\ G' : \mathbf{SP}\ \tau\ \tau'$ as required.

Case (*Out*). *We have the derivation:*

$$\emptyset \vdash \mathbf{PutSP}\ E_1\ E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!E_1} E_2$$

From (*Comp Put*) *there is some* H_1 *and* H_2 *such that* $H = \mathbf{PutSP}\ H_1\ H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \tau'$ *and* $\emptyset \vdash E_1 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *By the transition rule* (*Out*) *we infer the required derivation* $\emptyset \vdash \mathbf{PutSP}\ H_1\ H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{!H_1}{\Longrightarrow} H_2$.

Case ($Feed_1$). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?E_2} E_3}{\emptyset \vdash E_1 \ll E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\nu} E_3}$$

From the (*Comp Feed*) *rule there is some* H_1 *and* H_2 *such that* $H = H_1 \ll H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau$. *By induction for all* G' *such that* $\emptyset \vdash E_2 \approx^{h^\bullet} G' : \tau$ *then* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *We can instantiate* G' *to be* H_2 *yielding the transition* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?H_2}}{\Longrightarrow} H_3$, *and from the* ($Feed_1$) *transition rule we can infer* $\emptyset \vdash H_1 \ll H_2 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3$.

Case ($Feed_2$). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_3}{\emptyset \vdash E_1 \ll E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_3 \ll E_2}$$

where $\alpha \in \{\nu, !G\}$. *From the* (*Comp Feed*) *rule there is some* H_1 *and* H_2 *such that* $H = H_1 \ll H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau$. *We consider each of the two cases for* α *in turn:*

- $\alpha = \nu$, *then by induction we have the derivation* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using the* ($Feed_2$) *rule with this we can infer* $\emptyset \vdash H_1 \ll H_2 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3 \ll H_2$.
- $\alpha = !G$, *then by induction for some* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ *then* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using the* ($Feed_2$) *rule with this we can infer* $\emptyset \vdash H_1 \ll H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_3 \ll H_2$.

Finally, in both cases we can use the (*Comp Feed*) *and* (*Cand Cong*) *rules to obtain* $\emptyset \vdash E_3 \ll E_2 \approx^{h^\bullet} H_3 \ll H_2 : \mathbf{SP}\ \tau\ \tau'$ *as required.*

Case (Ser_1). *We have the derivation:*

$$\frac{\emptyset \vdash E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_3}{\emptyset \vdash E_1 >==< E_2 : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\alpha} E_1 >==< E_3}$$

where $\alpha \in \{\nu, ?G\}$. *From the* (*Comp Ser*) *rule then there is some* H_1 *and* H_2 *such that* $H = H_1 >==< H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau'\ \tau''$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *We consider each of the two cases for* α *in turn:*

- $\alpha = \nu$, *then by induction we have the derivation* $\emptyset \vdash H_2 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *From the* ($Ser_1$) *rule we can infer* $\emptyset \vdash H_1 >==< H_2 : \mathbf{SP}\ \tau\ \tau'' \Longrightarrow H_1 >==< H_3$.
- $\alpha = ?G$, *then by induction for all* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ *then* $\emptyset \vdash H_2 \stackrel{\widehat{?G'}}{\Longrightarrow} H_3 : \mathbf{SP}\ \tau\ \tau'$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using the* ($Ser_1$) *rule we can thus infer* $\emptyset \vdash H_1 >==< H_2 : \mathbf{SP}\ \tau\ \tau'' \stackrel{\widehat{?G'}}{\Longrightarrow} H_1 >==< H_3$.

Finally, in both cases we can use the (*Comp Ser*) *and* (*Cand Cong*) *rules to obtain* $\emptyset \vdash E_1 >==< E_3 \approx^{h^\bullet} H_1 >==< H_3 : \mathbf{SP}\ \tau\ \tau''$ *as required.*

Case (Ser_2). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_3}{\emptyset \vdash E_1 >==< E_2 : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\alpha} E_3 >==< E_2}$$

where $\alpha \in \{\nu, !G\}$. *From the* (*Comp Ser*) *rule then there is some* H_1 *and* H_2 *such that* $H = H_1 >==< H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau'\ \tau''$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *We consider each of the two cases for* α *in turn:*

- $\alpha = \nu$, *then by induction we have the derivation* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *From the* ($Ser_1$) *rule we can infer* $\emptyset \vdash H_1 >==< H_2 : \mathbf{SP}\ \tau\ \tau'' \Longrightarrow H_3 >==< H_1$.
- $\alpha = ?G$, *then by induction for all* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ *then* $\emptyset \vdash H_1 \stackrel{\widehat{?G'}}{\Longrightarrow} H_3 : \mathbf{SP}\ \tau\ \tau'$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using the* ($Ser_1$) *rule we can thus infer* $\emptyset \vdash H_1 >==< H_2 : \mathbf{SP}\ \tau\ \tau'' \stackrel{\widehat{?G'}}{\Longrightarrow} H_3 >==< H_2$.

Finally, in both cases we can use the (*Comp Ser*) *and* (*Cand Cong*) *rules to obtain* $\emptyset \vdash E_3 >==< E_2 \approx^{h^\bullet} H_3 >==< H_2 : \mathbf{SP}\ \tau\ \tau''$ *as required.*

Case (Ser_3). *We have the derivation:*

$$\frac{\emptyset \vdash E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G} E_3}{\emptyset \vdash E_1 >==< E_2 : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu} E_1 \ll G >==< E_3}$$

From ($Comp\ Ser$) *there is some* H_1 *and* H_2 *such that* $H = H_1 >==< H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau'\ \tau''$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *By induction then for some* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ *then* $\emptyset \vdash H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *From the* ($Ser_1$) *and* ($Ser_3$) *rules we can thus deduce the derivation* $\emptyset \vdash H_1 >==< H_2 : \mathbf{SP}\ \tau\ \tau'' \Longrightarrow H_1 \ll G' >==< H_3$. *Since* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau'\ \tau''$ *and* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ *then by using* ($Comp\ Feed$) *and* ($Cand\ \ Cong$) *we obtain* $\emptyset \vdash E_1 \ll G \approx^{h^\bullet} H_1 \ll G' : \mathbf{SP}\ \tau'\ \tau''$. *From this using the* ($Comp\ Ser$) *and* ($Cand\ \ Cong$) *rules we obtain the required derivation* $\emptyset \vdash E_1 \ll G >==< E_3 \approx^{h^\bullet} H_1 \ll G' >==< H_3 : \mathbf{SP}\ \tau\ \tau''$.

Case (Par_1). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_3}{\emptyset \vdash E_1 >*< E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_3 >*< E_2}$$

where $\alpha \in \{\nu, !G\}$. *From* ($Comp\ Par$) *there is some* H_1 *and* H_2 *such that* $H = H_1 >*< H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *We consider each of the two cases for* α *in turn:*

- $\alpha = \nu$, *then by induction we have the derivation* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *From the* ($Par_1$) *rule we obtain the derivation* $\emptyset \vdash H_1 >*< H_2 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3 >*< H_2$.
- $\alpha = !G$, *then by induction for some* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ *then* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using this with the* (Par_1) *we get* $\emptyset \vdash H_1 >*< H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_3 >*< H_2$.

Finally, because $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$ *in both cases then we can use the* ($Comp\ Par$) *and* ($Cand\ \ Cong$) *rules to obtain* $\emptyset \vdash E_3 >*< E_2 \approx^{h^\bullet} H_3 >*< H_2 : \mathbf{SP}\ \tau\ \tau'$ *as required.*

Case (Par_2). *We have the derivation:*

$$\frac{\emptyset \vdash E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_3}{\emptyset \vdash E_1 >\!*\!< E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_1 >\!*\!< E_3}$$

where $\alpha \in \{\nu, !G\}$. *From* (*Comp Par*) *there is some* H_1 *and* H_2 *such that* $H = H_1 >\!*\!< H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *We consider each of the two cases for* α *in turn:*

- $\alpha = \nu$, *then by induction we have the derivation* $\emptyset \vdash H_2 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *From the* ($Par_1$) *rule we obtain the derivation* $\emptyset \vdash H_1 >\!*\!< H_2 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_1 >\!*\!< H_3$.
- $\alpha = !G$, *then by induction for some* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ *then* $\emptyset \vdash H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using this with the* (Par_1) *we get* $\emptyset \vdash H_1 >\!*\!< H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_1 >\!*\!< H_3$.

Finally, because $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$ *in both cases then we can use the* (*Comp Par*) *and* (*Cand Cong*) *rules to obtain* $\emptyset \vdash E_1 >\!*\!< E_3 \approx^{h^\bullet} H_1 >\!*\!< H_3 : \mathbf{SP}\ \tau\ \tau'$ *as required.*

Case (Par_3). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E_3}{\emptyset \vdash E_1 >\!*\!< E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E_3 >\!*\!< (E_2 \ll G)}$$

From (*Comp Par*) *there is some* H_1 *and* H_2 *such that* $H = H_1 >\!*\!< H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *By induction then for all* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ *then* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using this along with the* (Par_1) *and* (Par_3) *rules we can infer that* $\emptyset \vdash H_1 >\!*\!< H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} H_3 >\!*\!< (H_2 \ll G')$. *We can infer* $\emptyset \vdash E_2 \ll G \approx^{h^\bullet} H_2 \ll G' : \mathbf{SP}\ \tau\ \tau'$ *from the* (*Comp Feed*) *and* (*Cand Cong*) *rules. Finally, from* (*Comp Par*) *and* (*Cand Cong*) *we obtain* $\emptyset \vdash E_3 >\!*\!< (E_2 \ll G) \approx^{h^\bullet} H_3 >\!*\!< (H_2 \ll G') : \mathbf{SP}\ \tau\ \tau'$ *as required.*

Case (Par_4). *We have the derivation:*

$$\frac{\emptyset \vdash E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E_3}{\emptyset \vdash E_1 >\!*\!< E_2 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} (E_1 \ll G) >\!*\!< E_3}$$

From ($Comp\ Par$) *there is some* H_1 *and* H_2 *such that* $H = H_1 >\!*\!< H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *By induction then for all* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ *then* $\emptyset \vdash H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{SP}\ \tau\ \tau'$. *Using this with* ($Par_1$) *and* ($Par_4$) *we obtain the derivation* $\emptyset \vdash H_1 >\!*\!< H_2 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} (H_1 \ll G') >\!*\!< H_3$. *We can show that* $\emptyset \vdash E_1 \ll G \approx^{h^\bullet} H_1 \ll G' : \mathbf{SP}\ \tau\ \tau'$ *using the* ($Comp\ Feed$) *and* ($Cand\ Cong$) *rules. Finally, using this with the* ($Comp\ Par$) *and* ($Cand\ Cong$) *rules we obtain* $\emptyset \vdash (E_1 \ll G) >\!*\!< E_3 \approx^{h^\bullet} (H_1 \ll G') >\!*\!< H_3 : \mathbf{SP}\ \tau\ \tau'$ *as required.*

Case ($Loop_1$). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau \xrightarrow{\alpha} E_2}{\emptyset \vdash \mathbf{LoopSP}\ E_1 : \mathbf{SP}\ \tau\ \tau \xrightarrow{\alpha} \mathbf{LoopSP}\ E_2}$$

where $\alpha \in \{\nu, ?G\}$. *From the* ($Comp\ Loop$) *rule there is some* H_1 *such that* $H = \mathbf{LoopSP}\ H_1$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau$. *We consider each of the two cases for* α *in turn:*

- $\alpha = \nu$, *then by induction we have the derivation* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau \Longrightarrow H_2$ *where* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau$. *From the* ($Loop_1$) *rule we can thus infer the derivation* $\emptyset \vdash \mathbf{LoopSP}\ H_1 : \mathbf{SP}\ \tau\ \tau \Longrightarrow \mathbf{LoopSP}\ H_2$.
- $\alpha = ?G$, *then by induction for all* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ *then* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau \stackrel{\widehat{?G'}}{\Longrightarrow} H_2$ *where* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau$. *Using the* ($Loop_1$) *rule we obtain* $\emptyset \vdash \mathbf{LoopSP}\ H_1 : \mathbf{SP}\ \tau\ \tau \stackrel{\widehat{?G'}}{\Longrightarrow} \mathbf{LoopSP}\ H_2$.

Finally, in both cases we can use the ($Comp\ Loop$) *and* ($Cand\ Cong$) *rules to obtain* $\emptyset \vdash \mathbf{LoopSP}\ E_2 \approx^{h^\bullet} \mathbf{LoopSP}\ H_2 : \mathbf{SP}\ \tau\ \tau$ *as required.*

Case ($Loop_2$). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau \xrightarrow{!G} E_2}{\emptyset \vdash \mathbf{LoopSP}\ E_1 : \mathbf{SP}\ \tau\ \tau \xrightarrow{!G} \mathbf{LoopSP}\ E_2 \ll G}$$

From (Comp Loop) there is some H_1 such that $H = \mathbf{LoopSP}\ H_1$ where $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau$. By induction for some G' such that $\emptyset \vdash G \approx^{h^\bullet} G' : \tau$ then $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau \stackrel{\widehat{!G'}}{\Longrightarrow} H_2$ where $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau$. From ($Loop_1$) and ($Loop_2$) we can derive $\emptyset \vdash \mathbf{LoopSP}\ H_1 : \mathbf{SP}\ \tau\ \tau \stackrel{\widehat{!G'}}{\Longrightarrow} \mathbf{LoopSP}\ H_2 \ll G'$. Since $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau$ and $\emptyset \vdash G \approx^{h^\bullet} G' : \mathbf{SP}\ \tau\ \tau$ then by using the (Comp Feed) and (Cand Cong) rules we get $\emptyset \vdash E_2 \ll G \approx^{h^\bullet} H_2 \ll G' : \mathbf{SP}\ \tau\ \tau$. Finally, using this with the (Comp Loop) and (Cand Cong) rules we obtain $\emptyset \vdash \mathbf{LoopSP}\ E_2 \ll G \approx^{h^\bullet} \mathbf{LoopSP}\ H_2 \ll G' : \mathbf{SP}\ \tau\ \tau$ as required.

Case (Dyn_1). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E_2}{\emptyset \vdash \mathbf{DynSP}\ E_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{\alpha} \mathbf{DynSP}\ E_2}$$

where $\alpha \in \{\nu, !G\}$. From the (Comp Dyn) rule there is some H_1 such that $H = \mathbf{DynSP}\ H_1$ where $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$. We consider each of the two cases for α in turn:

- *$\alpha = \nu$, then by induction we have the derivation $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \Longrightarrow H_2$ where $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. From the ($Dyn_1$) rule we thus obtain $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \Longrightarrow \mathbf{DynSP}\ H_2$.*
- *$\alpha = !G$, then by induction for some G' such that $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ then $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} H_2$ $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. Using the ($Dyn_1$) rule yields $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{!G'}}{\Longrightarrow} \mathbf{DynSP}\ H_2$.*

Finally, because $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$ then we can use (Comp Dyn) and (Cand Cong) to obtain $\emptyset \vdash \mathbf{DynSP}\ E_2 \approx^{h^\bullet} \mathbf{DynSP}\ H_2 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'$ as required.

Case (Dyn_2). *We have the derivation:*

$$\frac{\emptyset \vdash G : \mathbf{SP}\ \tau\ \tau' + \tau \xrightarrow{\nu} G' \quad \emptyset \vdash \mathbf{DynSP}\ E_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\,G'} E_2}{\emptyset \vdash \mathbf{DynSP}\ E_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \xrightarrow{?\,G} E_2}$$

From (Comp Dyn) there must be some H_1 such that $H = \mathbf{DynSP}\ H_1$ where $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$. By induction for all G'' such that $\emptyset \vdash G \approx^{h^\bullet} G'' : \mathbf{SP}\ \tau\ \tau' + \tau$ then $\emptyset \vdash G'' : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G'''$ where $\emptyset \vdash G' \approx^{h^\bullet} G''' :$

$\mathbf{SP}\ \tau\ \tau' + \tau$. *Also by induction for all* G'''' *such that* $\emptyset \vdash G' \approx^{h^\bullet} G'''' : \mathbf{SP}\ \tau\ \tau' + \tau$ *then* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G''''}}{\Longrightarrow} H_2$ *where* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'$. *Instantiating* G'''' *to be* G''' *we have the transition* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G'''}}{\Longrightarrow} H_2$. *Using this with* $\emptyset \vdash G'' : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow G'''$ *and Lemma A.5 then we obtain* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G''}}{\Longrightarrow} H_2$ *as required.*

Case (Dyn_3). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{?G}{\longrightarrow} E_2}{\emptyset \vdash \mathbf{DynSP}\ E_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{?\mathbf{Right}\ G}{\longrightarrow} \mathbf{DynSP}\ E_2}$$

From (*Comp Dyn*) *there must be some* H_1 *such that* $H = \mathbf{DynSP}\ H_1$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$. *For all* G' *where* $\emptyset \vdash \mathbf{Right}\ G \approx^{h^\bullet} G' : \tau$ *we need to show that* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} H_3$ *and* $\emptyset \vdash \mathbf{DynSP}\ E_2 \approx^{h^\bullet} H_3 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'$. *From Lemma A.3 there is some* G'' *such that* $\emptyset \vdash G' : \tau \Longrightarrow \mathbf{Right}\ G''$ *and* $\emptyset \vdash G \approx^{h^\bullet} G'' : \tau$. *By induction for all* G''' *such that* $\emptyset \vdash G \approx^{h^\bullet} G''' : \tau$ *then* $\emptyset \vdash H_1 : \mathbf{SP}\ \tau\ \tau' \stackrel{\widehat{?G'''}}{\Longrightarrow} H_2$ *where* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$. *Using this with* ($Dyn_1$) *and* ($Dyn_3$) *we obtain that* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?\mathbf{Right}}\ G'''}{\Longrightarrow} \mathbf{DynSP}\ H_2$. *Instantiating* G''' *to be* G'' *gives* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?\mathbf{Right}}\ G''}{\Longrightarrow} \mathbf{DynSP}\ H_2$. *Using this and* $\emptyset \vdash G' : \tau \Longrightarrow \mathbf{Right}\ G''$ *with Lemma A.5 then we can show that* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} \mathbf{DynSP}\ H_2$. *Finally, because* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau\ \tau'$ *then by using* (*Comp Dyn*) *and* (*Cand Cong*) *we obtain* $\emptyset \vdash \mathbf{DynSP}\ E_2 \approx^{h^\bullet} \mathbf{DynSP}\ H_2 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'$ *as required.*

Case (Dyn_4). *We have the derivation:*

$$\frac{\emptyset \vdash ?G : \mathbf{SP}\ \tau\ \tau'}{\emptyset \vdash \mathbf{DynSP}\ E_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{?\mathbf{Left}\ G}{\longrightarrow} \mathbf{DynSP}\ G}$$

From (*Comp Dyn*) *there must be some* H_1 *such that* $H = \mathbf{DynSP}\ H_1$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{SP}\ \tau\ \tau'$. *For all* G' *where* $\emptyset \vdash \mathbf{Left}\ G \approx^{h^\bullet} G' : SP\tau\tau' + \tau$ *we need to show that* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?G'}}{\Longrightarrow} H_3$ *and* $\emptyset \vdash \mathbf{DynSP}\ G \approx^{h^\bullet} H_3 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'$. *From Lemma A.3 there is some* G'' *such that* $\emptyset \vdash G' : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow \mathbf{Left}\ G''$ *and* $\emptyset \vdash G \approx^{h^\bullet} G'' :$

$\mathbf{SP}\ \tau\ \tau'$. *Using the* (Dyn_4) *transition rule we can infer that for all* G''' *such that* $\emptyset \vdash G \approx^{h^\bullet} G''' : \mathbf{SP}\ \tau\ \tau'$ *then* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{\mathbf{Left}\ ?}}{\Longrightarrow} \mathbf{DynSP}\ G'''$. *Instantiating* G''' *to be* G'' *we obtain the transition* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{\mathbf{Left}\ ?}}{\Longrightarrow} \mathbf{DynSP}\ G''$. *Using this with* $\emptyset \vdash G' : \mathbf{SP}\ \tau\ \tau' + \tau \Longrightarrow \mathbf{Left}\ G''$ *and Lemma A.5 then we can show that* $\emptyset \vdash \mathbf{DynSP}\ H_1 : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau' \stackrel{\widehat{?\,G'}}{\Longrightarrow} \mathbf{DynSP}\ G''$. *Finally, from* $(Comp\ Dyn)$ *and* $(Cand\ Cong)$ *we obtain* $\emptyset \vdash \mathbf{DynSP}\ G \approx^{h^\bullet} \mathbf{DynSP}\ G'' : \mathbf{SP}\ (\mathbf{SP}\ \tau\ \tau' + \tau)\ \tau'$ *as required.*

Case (Rec). *We have the derivation:*

$$\frac{\emptyset \vdash E_1[(\mathbf{Fix}\ x.E_1)/x] : \tau \xrightarrow{\alpha} E_2}{\emptyset \vdash \mathbf{Fix}\ x.E_1 : \tau \xrightarrow{\alpha} E_2}$$

From the $(Comp\ Fix)$ *rule there is some* H_1 *such that* $H = \mathbf{Fix}\ x.H_1$ *where* $x : \tau \vdash E_1 \approx^{h^\bullet} H_1 : \tau$. *Using the* $(Comp\ Fix)$ *and* $(Cand\ Cong)$ *rules we can show that* $\emptyset \vdash \mathbf{Fix}\ x.E_1 \approx^{h^\bullet} \mathbf{Fix}\ x.H_1 : \tau$. *We can now use the* $(Cand\ Subst)$ *rule to show that* $\emptyset \vdash E_1[(\mathbf{Fix}\ x.E_1)/x] \approx^{h^\bullet} H_1[(\mathbf{Fix}\ x.H_1)/x] : \tau$ *and proceed by considering each of the possibilities for* α *in turn:*

- $\alpha = \nu$, *then by induction we have that* $\emptyset \vdash H_1[(\mathbf{Fix}\ x.H_1)/x] : \tau \Longrightarrow H_2$ *where* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau$. *From the* (Rec) *rule we can thus infer the derivation* $\emptyset \vdash \mathbf{Fix}\ x.H_1 : \tau \Longrightarrow H_2$.
- $\alpha = ?G$, *then by induction for all* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau'$ *then* $\emptyset \vdash H_1[(\mathbf{Fix}\ x.H_1)/x] : \mathbf{SP}\ \tau'\ \tau'' \stackrel{\widehat{?\,G'}}{\Longrightarrow} H_2$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau'\ \tau''$. *From the* (Rec) *rule we can thus infer* $\emptyset \vdash \mathbf{Fix}\ x.H_1 : \tau \stackrel{\widehat{?\,G'}}{\Longrightarrow} H_2$.
- $\alpha = !G$, *then by induction for some* G' *such that* $\emptyset \vdash G \approx^{h^\bullet} G' : \tau''$ *then* $\emptyset \vdash H_1[(\mathbf{Fix}\ x.H_1)/x] : \mathbf{SP}\ \tau'\ \tau'' \stackrel{\widehat{!\,G'}}{\Longrightarrow} H_2$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{SP}\ \tau'\ \tau''$. *From the* (Rec) *rule we can thus infer* $\emptyset \vdash \mathbf{Fix}\ x.H_1 : \tau \stackrel{\widehat{!\,G'}}{\Longrightarrow} H_2$.

Case (Lam_1). *We have the derivation:*

$$\emptyset \vdash (\lambda x.E_1)\ E_2 : \tau \xrightarrow{\nu} E_1[E_2/x]$$

From $(Comp\ Abs)$ *there is some* H_1 *and* H_2 *such that* $H = H_1\ H_2$ *where* $\emptyset \vdash (\lambda x.E_1) \approx^{h^\bullet} H_1 : \tau' \to \tau$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau'$. *From Lemma A.3 there is*

some H_3 such that $\emptyset \vdash H_1 : \tau' \to \tau \Longrightarrow (\lambda x.H_3)$ where $x : \tau \vdash E_1 \approx^{h^\bullet} H_3 : \tau$. Using the (Lam_2) rule we can thus derive that $\emptyset \vdash H_1\ H_2 : \tau \Longrightarrow (\lambda x.H_3)\ H_2$. From this and (Lam_1) we obtain the derivation $\emptyset \vdash H_1\ H_2 : \tau \Longrightarrow H_3[H_2/x]$. Since $x : \tau \vdash E_1 \approx^{h^\bullet} H_3 : \tau$ and $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau$ then by using (Cand Subst) we get $\emptyset \vdash E_1[E_2/x] \approx^{h^\bullet} H_3[H_2/x] : \tau$ as required.

Case (Lam_2). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \tau \to \tau' \xrightarrow{\nu} E_3}{\emptyset \vdash E_1\ E_2 : \tau' \xrightarrow{\nu} E_3\ E_2}$$

From the (Comp App) rule there is some H_1 and H_2 such that $H = H_1\ H_2$ where $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \tau \to \tau'$ and $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau'$. By induction we have the derivation $\emptyset \vdash H_1 : \tau \to \tau' \Longrightarrow H_3$ where $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \tau \to \tau'$. Using the (Lam_2) rule we can derive $\emptyset \vdash H_1\ H_2 : \tau' \Longrightarrow H_3\ H_2$. Since $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \tau \to \tau'$ and $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau'$ then by using the (Comp App) and (Cand Cong) rules we obtain $\emptyset \vdash E_3\ E_2 \approx^{h^\bullet} H_3\ H_2 : \tau'$ as required.

Case $(Case_1)$. *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \tau + \tau' \xrightarrow{\nu} E_4}{\emptyset \vdash \mathbf{Case}\ E_1 \to E_2, E_3 : \tau'' \xrightarrow{\nu} \mathbf{Case}\ E_4 \to E_2, E_3}$$

From the (Comp Case) rule there must be some H_1, H_2, and H_3 such that $H = \mathbf{Case}\ H_1 \to H_2, H_3$ where $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \tau + \tau'$, $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau''$ and $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \tau''$. By induction we have $\emptyset \vdash H_1 : \tau + \tau' \Longrightarrow H_4$ where $\emptyset \vdash E_4 \approx^{h^\bullet} H_4 : \tau + \tau'$. Using this with the $(Case_1)$ rule we obtain the derivation $\emptyset \vdash \mathbf{Case}\ H_1 \to H_2, H_3 : \tau'' \Longrightarrow \mathbf{Case}\ H_4 \to H_2, H_3$. From (Comp Case) and (Cand Cong) we can show $\emptyset \vdash \mathbf{Case}\ E_4 \to E_2, E_3 \approx^{h^\bullet} \mathbf{Case}\ H_4 \to H_2, H_3 : \tau''$ as required.

Case $(Case_2)$. *We have the derivation:*

$$\emptyset \vdash \mathbf{Case\ Left}\ E_1 \to E_2, E_3 : \tau'' \xrightarrow{\nu} E_2\ E_1$$

Using the (Comp Case) rule there must be some H_1, H_2, and H_3 such that $H = \mathbf{Case}\ H_1 \to H_2, H_3$ where $\emptyset \vdash \mathbf{Left}\ E_1 \approx^{h^\bullet} H_1 : \tau + \tau'$, $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau''$ and $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \tau''$. From Lemma A.3 then there must be some H_4 such

that $\emptyset \vdash H_1 : \tau + \tau' \Longrightarrow \mathbf{Left}\ H_4$ *and* $\emptyset \vdash E_1 \approx^{h^\bullet} H_4 : \tau$. *Using this with the* ($Case_1$) *rule yields* $\emptyset \vdash \mathbf{Case}\ H_1 \to H_2, H_3 : \tau'' \Longrightarrow \mathbf{Case\ Left}\ H_4 \to H_2, H_3$. *From this and* ($Case_2$) *we can show* $\emptyset \vdash \mathbf{Case}\ H_1 \to H_2, H_3 : \tau'' \Longrightarrow H_2\ H_4$. *Finally,* (*Comp App*) *and* (*Cand Cong*) *yields* $\emptyset \vdash E_2\ E_1 \approx^{h^\bullet} H_2\ H_4 : \tau''$ *as required.*

Case ($Case_3$). *We have the derivation:*

$$\emptyset \vdash \mathbf{Case\ Right}\ E_1 \to E_2, E_3 : \tau'' \xrightarrow{\nu} E_2\ E_1$$

Using the (*Comp Case*) *rule there must be some* H_1, H_2, *and* H_3 *such that* $H = \mathbf{Case}\ H_1 \to H_2, H_3$ *where* $\emptyset \vdash \mathbf{Right}\ E_1 \approx^{h^\bullet} H_1 : \tau + \tau'$, $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau''$ *and* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \tau''$. *From Lemma A.3 then there must be some* H_4 *such that* $\emptyset \vdash H_1 : \tau + \tau' \Longrightarrow \mathbf{Right}\ H_4$ *and* $\emptyset \vdash E_1 \approx^{h^\bullet} H_4 : \tau'$. *Using this with* ($Case_1$) *yields* $\emptyset \vdash \mathbf{Case}\ H_1 \to H_2, H_3 : \tau'' \Longrightarrow \mathbf{Case\ Right}\ H_4 \to H_2, H_3$. *From this and* ($Case_2$) *we get* $\emptyset \vdash \mathbf{Case}\ H_1 \to H_2, H_3 : \tau'' \Longrightarrow H_2\ H_4$. *Finally, from the* (*Comp App*) *and* (*Cand Cong*) *rules we can show* $\emptyset \vdash E_2\ E_1 \approx^{h^\bullet} H_2\ H_4 : \tau''$ *as required.*

Case ($Cond_1$). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{bool} \xrightarrow{\nu} E_4}{\emptyset \vdash \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : \tau \xrightarrow{\nu} \mathbf{if}\ E_4\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3}$$

From the (*Comp If*) *rule there must be some* H_1, H_2, *and* H_3 *such that* $H = \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3$ *with* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{bool}$, $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \tau$ *and* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \tau$. *Induction gives* $\emptyset \vdash H_1 : \mathbf{bool} \Longrightarrow H_4$ *where* $\emptyset \vdash E_4 \approx^{h^\bullet} H_4 : \mathbf{bool}$. *Using the* ($Cond_1$) *rule we obtain the derivation* $\emptyset \vdash \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3 : \tau \Longrightarrow \mathbf{if}\ H_4\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3$. *Finally, by using the* (*Comp If*) *and* (*Cand Cong*) *rules results in the derivation* $\emptyset \vdash \mathbf{if}\ E_4\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 \approx^{h^\bullet} \mathbf{if}\ H_4\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3 : \tau$ *as required.*

Case ($Cond_2$). *We have the derivation:*

$$\emptyset \vdash \mathbf{if\ true\ then}\ E_1\ \mathbf{else}\ E_2 : \tau \xrightarrow{\nu} E_1$$

From the (*Comp If*) *rule there must be some* H_1, H_2, *and* H_3 *such that* $H = \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3$ *where* $\emptyset \vdash \mathbf{true} \approx^{h^\bullet} H_1 : \mathbf{bool}$, $\emptyset \vdash E_1 \approx^{h^\bullet} H_2 : \tau$ *and*

$\emptyset \vdash E_2 \approx^{h^\bullet} H_3 : \tau$. *From Lemma A.3 we have that* $\emptyset \vdash H_1 : \mathbf{bool} \Longrightarrow \mathbf{true}$. *Using this with the* $(Cond_1)$ *rule we can derive* $\emptyset \vdash \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3 : \tau \Longrightarrow \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3$, *and so from the* $(Cond_2)$ *rule we obtain the derivation* $\emptyset \vdash \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3 : \tau \Longrightarrow H_2$ *as required.*

Case $(Cond_3)$. *We have the derivation:*

$$\emptyset \vdash \mathbf{if}\ \mathbf{false}\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 : \tau \xrightarrow{\nu} E_2$$

From the $(Comp\ If)$ *rule there must be some* H_1, H_2, *and* H_3 *such that* $H = \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3$ *where* $\emptyset \vdash \mathbf{true} \approx^{h^\bullet} H_1 : \mathbf{bool}$, $\emptyset \vdash E_1 \approx^{h^\bullet} H_2 : \tau$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_3 : \tau$. *From Lemma A.3 we have that* $\emptyset \vdash H_1 : \mathbf{bool} \Longrightarrow \mathbf{false}$. *Using this with the* $(Cond_1)$ *rule we can derive* $\emptyset \vdash \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3 : \tau \Longrightarrow \mathbf{if}\ \mathbf{false}\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3$, *and so from the* $(Cond_2)$ *rule we obtain the derivation* $\emptyset \vdash \mathbf{if}\ H_1\ \mathbf{then}\ H_2\ \mathbf{else}\ H_3 : \tau \Longrightarrow H_3$ *as required.*

Case (Add_1). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{num} \xrightarrow{\nu} E_3}{\emptyset \vdash \mathbf{add}\ E_1\ E_2 : \mathbf{num} \xrightarrow{\nu} \mathbf{add}\ E_3\ E_2}$$

Using the $(Comp\ Add)$ *rule there must be some* H_1 *and* H_2 *such that* $H = \mathbf{add}\ H_1\ H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{num}$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{num}$. *By induction we have that* $\emptyset \vdash H_1 : \mathbf{num} \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{num}$. *Using* (Add_1) *yields* $\emptyset \vdash \mathbf{add}\ H_1\ H_2 : \mathbf{num} \Longrightarrow \mathbf{add}\ H_3\ H_2$. *Finally, using the* $(Comp\ Add)$ *and* $(Cand\ Cong)$ *rules we obtain the derivation* $\emptyset \vdash \mathbf{add}\ E_3\ E_2 \approx^{h^\bullet} \mathbf{add}\ H_3\ H_2 : \mathbf{num}$ *as required.*

Case (Add_2). *We have the derivation:*

$$\frac{\emptyset \vdash E_2 : \mathbf{num} \xrightarrow{\nu} E_3}{\emptyset \vdash \mathbf{add}\ E_1\ E_2 : \mathbf{num} \xrightarrow{\nu} \mathbf{add}\ E_1\ E_3}$$

From the $(Comp\ Add)$ *rule there is some* H_1 *and* H_2 *such that* $H = \mathbf{add}\ H_1\ H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{num}$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{num}$. *By induction we have the derivation* $\emptyset \vdash H_2 : \mathbf{num} \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{num}$. *Using the* (Add_1) *rule yields* $\emptyset \vdash \mathbf{add}\ H_1\ H_2 : \mathbf{num} \Longrightarrow \mathbf{add}\ H_1\ H_3$. *Finally, using the* $(Comp\ Add)$ *and* $(Cand\ Cong)$ *rules we obtain the derivation*

$\emptyset \vdash \mathbf{add}\ E_1\ E_3 \approx^{h^\bullet} \mathbf{add}\ H_1\ H_3 : \mathbf{num}$ *as required.*

Case (Add_3). *We have the derivation:*

$$\emptyset \vdash \mathbf{add}\ \mathbf{n}\ \mathbf{m} : \mathbf{num} \xrightarrow{\nu} \mathbf{n} + \mathbf{m}$$

From the (*Comp Add*) *rule there is some* H_1 *and* H_2 *such that* $H = \mathbf{add}\ H_1\ H_2$ *where* $\emptyset \vdash \mathbf{n} \approx^{h^\bullet} H_1 : \mathbf{num}$ *and* $\emptyset \vdash \mathbf{m} \approx^{h^\bullet} H_2 : \mathbf{num}$. *From Lemma A.3 then* $\emptyset \vdash H_1 : \mathbf{num} \Longrightarrow \mathbf{n}$ *and* $\emptyset \vdash H_2 : \mathbf{num} \Longrightarrow \mathbf{m}$. *Combining these transitions using the* (Add_1) *and* (Add_2) *rules we obtain the derivation* $\emptyset \vdash \mathbf{add}\ H_1\ H_2 : \mathbf{num} \Longrightarrow \mathbf{add}\ \mathbf{n}\ \mathbf{m}$. *Now using the* ($Add_3$) *rule we can derive that* $\emptyset \vdash \mathbf{add}\ H_1\ H_2 : \mathbf{num} \Longrightarrow \mathbf{n} + \mathbf{m}$. *Finally, from the* (*Comp Lit*) *and* (*Cand Cong*) *rules we can show that* $\emptyset \vdash \mathbf{n} + \mathbf{m} \approx^{h^\bullet} \mathbf{n} + \mathbf{m} : \mathbf{num}$ *as required.*

Case ($Equal_1$). *We have the derivation:*

$$\frac{\emptyset \vdash E_1 : \mathbf{num} \xrightarrow{\nu} E_3}{\emptyset \vdash \mathbf{equal}\ E_1\ E_2 : \mathbf{bool} \xrightarrow{\nu} \mathbf{equal}\ E_3\ E_2}$$

Using the (*Comp Equal*) *rule there must be some* H_1 *and* H_2 *such that* $H = \mathbf{equal}\ H_1\ H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{num}$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{num}$. *By induction we have that* $\emptyset \vdash H_1 : \mathbf{num} \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{num}$. *Using* ($Equal_1$) *yields* $\emptyset \vdash \mathbf{equal}\ H_1\ H_2 : \mathbf{bool} \Longrightarrow \mathbf{equal}\ H_3\ H_2$. *Finally, using the* (*Comp Equal*) *and* (*Cand Cong*) *rules we obtain the derivation* $\emptyset \vdash \mathbf{equal}\ E_3\ E_2 \approx^{h^\bullet} \mathbf{equal}\ H_3\ H_2 : \mathbf{bool}$ *as required.*

Case ($Equal_2$). *We have the derivation:*

$$\frac{\emptyset \vdash E_2 : \mathbf{num} \xrightarrow{\nu} E_3}{\emptyset \vdash \mathbf{equal}\ E_1\ E_2 : \mathbf{bool} \xrightarrow{\nu} \mathbf{equal}\ E_1\ E_3}$$

Using the (*Comp Equal*) *rule there must be some* H_1 *and* H_2 *such that* $H = \mathbf{equal}\ H_1\ H_2$ *where* $\emptyset \vdash E_1 \approx^{h^\bullet} H_1 : \mathbf{num}$ *and* $\emptyset \vdash E_2 \approx^{h^\bullet} H_2 : \mathbf{num}$. *By induction we have that* $\emptyset \vdash H_2 : \mathbf{num} \Longrightarrow H_3$ *where* $\emptyset \vdash E_3 \approx^{h^\bullet} H_3 : \mathbf{num}$. *Using* ($Equal_1$) *yields* $\emptyset \vdash \mathbf{equal}\ H_1\ H_2 : \mathbf{bool} \Longrightarrow \mathbf{equal}\ H_1\ H_3$. *Finally, using the* (*Comp Equal*) *and* (*Cand Cong*) *rules we obtain the derivation* $\emptyset \vdash \mathbf{equal}\ E_1\ E_3 \approx^{h^\bullet} \mathbf{equal}\ H_1\ H_3 : \mathbf{bool}$ *as required.*

Case (*Equal*$_3$). *We have the derivation:*

$$\emptyset \vdash \mathbf{equal\ n\ m} : \mathbf{bool} \xrightarrow{\nu} \mathbf{n} = \mathbf{m}$$

From the (*Comp Equal*) *rule there is some* H_1 *and* H_2 *such that* $H = \mathbf{equal}\ H_1\ H_2$ *where* $\emptyset \vdash \mathbf{n} \approx^{h^\bullet} H_1 : \mathbf{num}$ *and* $\emptyset \vdash \mathbf{m} \approx^{h^\bullet} H_2 : \mathbf{num}$. *From Lemma A.3 then* $\emptyset \vdash H_1 : \mathbf{num} \Longrightarrow \mathbf{n}$ *and* $\emptyset \vdash H_2 : \mathbf{num} \Longrightarrow \mathbf{m}$. *Combining these transitions using the* (*Equal*$_1$) *and* (*Equal*$_2$) *rules we obtain the derivation* $\emptyset \vdash \mathbf{equal}\ H_1\ H_2 : \mathbf{bool} \Longrightarrow \mathbf{equal\ n\ m}$. *Now using the* (*Equal*$_3$) *rule we can derive that* $\emptyset \vdash \mathbf{equal}\ H_1\ H_2 : \mathbf{bool} \Longrightarrow \mathbf{n} = \mathbf{m}$. *Finally, from the* (*Comp Lit*) *and* (*Cand Cong*) *rules we can show that* $\emptyset \vdash \mathbf{n} = \mathbf{m} \approx^{h^\bullet} \mathbf{n} = \mathbf{m} : \mathbf{bool}$ *as required.*

■

From the two lemmas A.3 and A.6 we can conclude that $\mathcal{S}$ is a weak higher-order simulation. However, we really need to show $\mathcal{S}$ to be a weak higher-order bisimulation. We proceed using an observation of Howe's [How96] that for a type-indexed relation $\mathcal{R}$ then the transitive reflexive closure of $\mathcal{R}^\bullet$ is symmetric:

Definition A.4 (Transitive reflexive closure) *The transitive reflexive closure of a type-indexed relation* $\mathcal{R}$ *is the least such relation satisfying the three inference rules in Figure A.5.*

$$(Incl) \quad \frac{, \vdash E\mathcal{R}F : \tau}{, \vdash E\mathcal{R}^*F : \tau}$$

$$(Refl) \quad , \vdash E\mathcal{R}^*E : \tau$$

$$(Trans) \quad \frac{, \vdash E\mathcal{R}^*F : \tau \quad , \vdash F\mathcal{R}G : \tau}{, \vdash E\mathcal{R}^*G : \tau}$$

Figure A.5: Transitive reflexive closure

Lemma A.7 *For any type-indexed equivalence relation,* $\mathcal{R}$ *then* $, \vdash E\mathcal{R}^\bullet F : \tau$ *implies* $, \vdash F\mathcal{R}^{\bullet *}E : \tau$.

Proof. *Proceeding by structural induction on E, we know from (Cand Def) that there is a G such that , $\vdash E\widehat{\mathcal{R}^\bullet}G : \tau$ and , $\vdash G\mathcal{R}^\circ F : \tau$. We illustrate the proof by examining two cases:*

Case *$E = x$. From the definition of compatible refinement, $G = x$, and from the symmetry of $\mathcal{R}^\circ$ we have that , $\vdash F\mathcal{R}^\circ x : \tau$. Using (Cand Sim) we obtain , $\vdash F\mathcal{R}^\bullet x : \tau$, and by the (Incl) rule we can conclude , $\vdash F\mathcal{R}^{\bullet *} x : \tau$ as required.*

Case *$E = \lambda x.E'$. From the definition of compatible refinement, $G = \lambda x.G'$, where , $\vdash E'\mathcal{R}^\bullet G' : \tau'$. By induction we can infer , $\vdash G'\mathcal{R}^{\bullet *} E' : \tau'$, and using the (Comp Abs) and (Cand Cong) rules we obtain , $\vdash \lambda x.G'\mathcal{R}^{\bullet *}\lambda x.E' : \tau \to \tau'$. From the symmetry of $\mathcal{R}^\circ$ we have that , $\vdash F\mathcal{R}^\circ \lambda x.G' : \tau \to \tau'$, and thus from (Cand Sim) we obtain , $\vdash F\mathcal{R}^\bullet \lambda x.G' : \tau \to \tau'$. Finally, by using the (Incl) and (Trans) rules we can conclude , $\vdash F\mathcal{R}^{\bullet *}\lambda x.E' : \tau \to \tau'$ as required.*

The other cases follow similarly. ∎

Lemma A.8 *If $\mathcal{R}$ is a type-indexed equivalence relation then $\mathcal{R}^{\bullet *}$ is symmetric.*

Proof. *The proof proceeds by induction on the derivation of , $\vdash E\mathcal{R}^{\bullet *}F : \tau$, and there are three cases to consider:*

Case *(Incl). Since , $\vdash E\mathcal{R}F : \tau$ then by Lemma A.7 we obtain , $\vdash F\mathcal{R}^* E : \tau$.*

Case *(Refl). In this case $F = E$, and from (Refl) we obtain , $\vdash F\mathcal{R}^{\bullet *} E : \tau$.*

Case *(Trans). In this case there must be some G such that , $\vdash E\mathcal{R}^* G : \tau$ and , $\vdash G\mathcal{R}^* F : \tau$. By induction we obtain the derivations , $\vdash G\mathcal{R}^* F : \tau$ and , $\vdash F\mathcal{R}^* G : \tau$, and so by using (Trans) we obtain , $\vdash F\mathcal{R}^* E : \tau$ as desired.*

∎

Lemma A.9 *The relation $\mathcal{S}^*$ is a weak higher-order simulation.*

Proof. *The proof follows by induction on the derivation of $\emptyset \vdash E\mathcal{S}^* F : \tau$, and in each case if E has a transition then we show the appropriate properties for $\mathcal{S}^*$ to be a weak higher-order simulation. There are three cases to consider:*

Case (*Incl*). *In this case it must be that* $\emptyset \vdash E\mathcal{S}F : \tau$ *and because* $\mathcal{S}$ *is a weak higher-order simulation then we are done.*

Case (*Refl*). *In this case* $E = F$*, and because the identity relation is a weak higher-order simulation then the result follows.*

Case (*Trans*). *From the* (*Trans*) *rule there is some* G *such that* $\emptyset \vdash E\mathcal{S}^* G : \tau$ *and* $\emptyset \vdash G\mathcal{S}^* F : \tau$*. By induction we know that if* $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ *then there must be some* G' *and* α' *such that* $\emptyset \vdash G : \tau \stackrel{\widehat{\alpha'}}{\Longrightarrow} G'$ *where* $\emptyset \vdash \alpha \mathcal{S}^{*\,a} \alpha' : \tau'$ *and* $\emptyset \vdash E'\mathcal{S}^* G' : \tau$*. Also by induction, because* $\emptyset \vdash G : \tau \stackrel{\widehat{\alpha'}}{\Longrightarrow} G'$ *then there must be some* F' *and* α'' *such that* $\emptyset \vdash F : \tau \stackrel{\widehat{\alpha''}}{\Longrightarrow} F'$ *where* $\emptyset \vdash \alpha' \mathcal{S}^{*\,a} \alpha'' : \tau'$ *and* $\emptyset \vdash G'\mathcal{S}^* F' : \tau$*. Finally, from the* (*Trans*) *rule we can derive that* $\emptyset \vdash E'\mathcal{S}^* F' : \tau$ *as required.* ∎

Lemma A.10 $\approx^{h^\bullet} \subseteq \mathcal{S}^\circ$.

Proof. *For any* , $\vdash E \approx^{h^\bullet} F : \tau$ *where* , $= \vec{x} : \vec{\tau}$*, then from repeated application of the* (*Cand Subst*) *rule it follows that for all* , *-closures* $\cdot[\vec{G}/\vec{x}]$ *where* $\emptyset \vdash \vec{G} : \vec{\tau}$ *that* $\emptyset \vdash E[\vec{G}/\vec{x}] \approx^{h^\bullet} F[\vec{G}/\vec{x}] : \tau$*. This is exactly the condition required in the definition of the open extension of a relation, and thus it follows that* $\emptyset \vdash E\mathcal{S}^\circ F : \tau$*.* ∎

We now have all the results to complete the proof, and begin by restating the theorem from Chapter 8:

Theorem 8.1 (Congruence) $\approx^{h^\circ}$ *is a congruence.*

Proof. *From Lemma A.8 it follows that* $\mathcal{S}^*$ *is symmetric, and using this with Lemma A.9 we conclude that* $\mathcal{S}^*$ *is a weak higher-order bisimulation, and thus* $\mathcal{S}^* \subseteq \approx^h$*. However, since* $\mathcal{S} \subseteq \mathcal{S}^*$ *by definition of transitive reflexive closure then we have that* $\mathcal{S} \subseteq \approx^h$*. From the monotonicity of open extension it follows that* $\mathcal{S}^\circ \subseteq \approx^{h^\circ}$*. Using Lemma A.10 we have that* $\approx^{h^\bullet} \subseteq \mathcal{S}^\circ$ *and so* $\approx^{h^\bullet} \subseteq \approx^{h^\circ}$*. Finally, combining this with* (*Cand Sim*) *we obtain* $\approx^{h^\bullet} = \approx^{h^\circ}$ *and since* $\approx^{h^\bullet}$ *contains its own compatible refinement then it must be a congruence and so* $\approx^{h^\circ}$ *is also a congruence.* ∎

References

[Abr89] Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[AGR88] E. Astesiano, A. Giovini, and G. Reggio. Generalized bisimulation in relational specification. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, volume 294 of *Lecture Notes in Computer Science*, pages 207–226. Springer-Verlag, 1988.

[BCP71] R.M. Burstall, J.S. Collins, and R.J. Popplestone. *Programming in POP-2.* Edinburgh University Press, 1971.

[BD77] R.M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BDD+93] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems - an introduction to FOCUS. Technical Report TUM-I9202, Technische Universität Müuchen, January 1993.

[Ber91] Thomas Berlage. *OSF/Motif: Concepts and Programming.* Addison-Wesley, 1991.

[BG88] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *The Science of Computer Programming*, 842, 1988.

[BMS80] R.M. Burstall, D.B. MacQueen, and D.T. Sanella. Hope: an experimental applicative language. Technical Report CSR-62-80, University of Edinburgh, 1980.

[Bou89] Gérard Boudol. Towards a lambda calculus for concurrent and communicating systems. In *TAPSOFT '89*, volume 351 of *LNCS*, pages 149–161, 1989.

[BS93] E. Barendsen and J.E.W Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In R. K. Shyamasundar, editor, *Proceedings of the Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 41–51. Springer-Verlag, LNCS 761, December 1993.

[Bur75] W.H. Burge. Stream processing functions. *IBM Journal of Research and Development*, 19:12–25, January 1975.

[Bur88] F.W. Burton. Nondeterminism with referential transparency in functional programming languages. *The Computer Journal*, 31:243–247, 1988.

[BvEG+87] Henk Barendregt, Marko van Eekelen, John Glauert, Richard Kennaway, Rinus Plasmeijer, and Ronan Sleep. Term graph rewriting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 141–158. Springer-Verlag, LNCS 259, 1987.

[CH93] Magnus Carlsson and Thomas Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, June 1993.

[CH95] Magnus Carlsson and Thomas Hallgren. The Fudget Library distribution. Available by anonymous FTP from `pub/haskell/chalmers` on `ftp.cs.chalmers.se`., 1995.

[CH98] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces.* Ph.D. dissertation, Department of Computing Science, Chalmers University of Technology, 1998.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. The Calculi of Lambda-Conversion. *Annals of Mathematics Studies*, 6, 1941.

[CVM97] Koen Claessen, Ton Vullinghs, and Erik Meijer. Structuring Graphical Paradigms in TkGofer. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 251–262, June 1997.

[Dan84] Roger B. Dannenberg. Arctic: A functional language for real-time control. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 96–103, August 1984.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, 1997.

[Fai85] Jon Fairbairn. Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus. Technical Report 75, University of Cambridge, May 1985.

[FGJ96] Sigbjørn Finne, Andrew D. Gordon, and Simon L. Peyton Jones. Concurrent Haskell. In *Proceedings of the Twenty Third ACM Symposium on Principles of Programming Languages (POPL)*, pages 295–308, January 1996.

[FHJ95] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. Computer Science Technical Report 95:05, School of Cognitive and Computing Sciences, University of Sussex, 1995.

[FHJ96] W. Ferreira, M. Hennessy, and A. Jeffrey. Combining the typed lambda-calculus with CCS. Computer Science Technical Report 2/96, University of Sussex, 1996.

[FJ95] Sigbjørn Finne and Simon L. Peyton Jones. Composing Haggis. In *Eurographics Workshop on Programming Paradigms in Computer Graphics*, April 1995.

[GGB87] T. Guatier, P. Le Guernia, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. Techni-

cal report, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cédex, France, 1987.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[GMP90] A. Giacalone, P. Mishra, and S. Prasad. Operational and algebraic semantics for facile: A symmetric integration of concurrent and functional programming. In *Proceedings of ICALP 90*. Springer-Verlag, LNCS 443, 1990.

[Gor92] Andrew D. Gordon. *Functional Programming and Input/Output*. Ph.D. dissertation, University of Cambridge, August 1992.

[Har87] D. Harrison. RUTH: A functional language for real-time programming. In G. Goos and J. Hartmanis, editors, *PARLE Parallel Architectures and Languages EUROPE*, pages 297–314. Springer-Verlag, LNCS 259, 1987.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 137–182. Springer Verlag, May 1995.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[How89] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[How96] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124:103–112, February 1996.

[HS89] P. Hudak and R. Sundaresh. On the expressiveness of purely functional i/o systems. Technical Report YALEU/DCS/RR-665, Yale University, Department of Computer Science, February 1989.

[Inc90] Sun Microsystems Inc. *OPEN LOOK Graphical User Interface Functional Specification.* Addison Wesley, January 1990.

[Jon91] Mark P. Jones. An Introduction to Gofer. Available as part of the standard Gofer distribution by anonymous FTP from `/nott-fp/languages/gofer` on `ftp.cs.nott.ac.uk`., 1991.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, May 1994.

[Jon95] Mark P. Jones. The Hugs distribution. Available by anonymous FTP from `nott-fp/languages/hugs` on `ftp.cs.nott.ac.uk`., 1995.

[Kah73] Gilles Kahn. A preliminary theory for parallel programs. Rapport de Recherche 6, IRIA, January 1973.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel processing. In *Proceedings of IFIP 74*, pages 471–475. North-Holland Publishing Company, 1974.

[Kar81] Kent Karlsson. Nebula: A functional operating system. Technical report, Programming Methodology Group, Chalmers University of Technology and University of Gothenburg, 1981.

[KM77] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. *Information Processing 77*, pages 993–998, 1977.

[KR89] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1989.

[Lan65] P.J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I and II. In *Communications of the ACM*, volume 8, pages 89–101,158–165, February and March 1965.

[Lan66] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[Lin97] Gary Shu Ling. Fran: Its semantics and existing problems. Available by HTTP from `/users/ling/ps/690Report.ps.zip` on `www.cs.yale.edu`., 1997.

[Mat93] H.F Mattson, Jr. *Discrete mathematics with applications*. John Wiley and Sons, Inc., 1993.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[Mil85] R. Milner. *Communication and Concurrency*. Prentice Hall, 1985.

[Mil90] R. Milner. Functions as processes. Research Report No. 1154, INRIA, February 1990.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, Asilomar, California, 1989.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report ECS-LFCS-89-85 and 86, Department of Computer Science, The University of Edinburgh, 1989.

[MT90] R. Milner and M. Tofte. Co-induction in relational semantics. In *Theoretical Computer Science*, volume 87, pages 209–220, September 1990.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Nob95] Rob Noble. *Lazy Functional Components for Graphical User Interfaces*. Ph.D. dissertation, Dept. of Computer Science, University of York, November 1995.

[NSvEP91] E. Nøcker, S. Smetsers, M. van Eekelen, and R. Plasmeijer. Concurrent clean. In Leeuwen Aarts and Rem, editors, *Proceedings of Parallel*

Architectures and Languages Europe, pages 202–219. Springer-Verlag, LNCS 505, 1991.

[OD93] Gerald K. Ostheimer and Antony J.T. Davie. π-calculus characterizations of some practical λ-calculus reduction strategies. Technical Report CS/93/14, Department of Mathematical and Computational Sciences, University of St. Andrews, October 1993.

[Ous94] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison Wesley, 1994.

[Par81] D.M. Park. Concurrency on automata and infinite sequences. In *Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[PH96] J. Peterson and K. Hammond. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language (Version 1.3). Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, May 1996.

[Pit94] A.M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[Plo75] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[Rep91] J.H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the 1991 ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[Rep92] J.H. Reppy. *Higher-order concurrency.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, January 1992.

[RS93] Alastair Reid and Satnam Singh. Implementing Fudgets with Standard Widget Sets. In *Glasgow functional programming workshop*, pages 222–235. Springer-Verlag, 1993.

[Sag97] Meurig Sage. Chronos users manual. Available by HTTP from `/~meurig/chronos/chronos.html` on `www.dcs.gla.ac.uk`., 1997.

[San92] Davide Sangiorgi. Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. Technical Report CST-99-93, Department of Computer Science, University of Edinburgh, 1992.

[San93] Davide Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS-LFCS-93-270, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1993.

[SB96] Enno Scholz and Boris Bokowski. PIDGETS++ – A C++ Framework Unifying Postscript Pictures, GUI Objects, and Lazy One-Way Constraints. In *Proceedings of the Twentieth International Conference on the Technology of Object-Oriented Languages and Systems*. Prentice-Hall, July 1996.

[SBvEP94] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In Schneider and Ehrig, editors, *Proceedings of Graph Transformations in Computers Science, International Workshop*, volume 776 of *Lecture Notes in Computer Science*, pages 358–379. Springer Verlag, 1994.

[Sch86] David A. Schmidt. *Denotational Semantics, A Methodology for Language Development*. Wm. C. Brown Publishers, 1986.

[Sch95] Enno Scholz. PIDGETS – Unifying Pictures and Widgets in a Simple Model for Concurrent Functional GUI Programming. Technical Report B 95-13, Free University of Berlin, 1995.

[Sew97] Peter Sewell. On implementations and semantics of a concurrent programming language. In Antoni Mazurkiewicz and Józef Winkowski, editors, *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405. Springer-Verlag, 1997.

[SG86] R.W. Scheiffler and J. Getty. The X Window System. In *ACM Transactions on Graphics*, volume 5, pages 79–109, April 1986.

[Spi90] Mike Spivey. A Functional Theory of Exceptions. *Science of Computer Programming*, 14:25–42, 1990.

[Str65] C. Strachey. A general purpose macrogenerator. *Computer Jrnl.*, 8:225–241, 1965.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison Wesley, 1997.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations – a mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974.

[Tay96] Colin J. Taylor. Embracing windows. Technical Report NOTTCS-TR-96-1, Department of Computer Science, University of Nottingham, November 1996.

[Tho87] Simon Thompson. Interactive functional programs: a method and a formal semantics. Technical Report 48, Computing Laboratory, University of Kent, Canterbury, UK, November 1987.

[Tho89] Bent Thomsen. A calculus of higher order communicating systems. In *Proceedings of the 16th ACM Symposium on the Principles of Programming Languages*, pages 143–154, 1989.

[Tho90] Bent Thomsen. *Calculi for Higher Order Communicating Systems.* Ph.D. dissertation, Department of Computing, Imperial College, 1990.

[Tur85] D.A. Turner. Miranda - a non strict functional language with polymorphic types. In *Proceedings IFIP Functional Programming Languages and Computer Architecture*, Nancy France, September 1985.

[Tur95] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* Ph.D. dissertation, University of Edinburgh, 1995.

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[VSS96] T. Vullinghs, W. Schulte, and T. Schwinn. An Introduction to Tk-Gofer. Technical Report Technical Report 96-03, University of Ulm, June 1996.

[WA85] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[Wad90] Philip Wadler. Comprehending Monads. In *ACM Conference on Lisp and Functional programming*, pages 61–78, Nice, France, June 1990.

[Wad92] Philip Wadler. The essence of functional programing (invited talk). In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–14, January 1992.