The Semantics and Implementation of Machine-Readable CSP

Bryan Scattergood

The Queen's College

Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford, Trinity 1998.

Oxford University Computing Laboratory

Programming Research Group

The Semantics and Implementation of Machine-Readable CSP

Bryan Scattergood

Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford, Trinity 1998.

Abstract

The CSP process-algebra provides a powerful mathematical notation for reasoning about concurrent systems, together with a rich supporting theory. This thesis adapts and extends CSP to produce a language which can be used with tools.

The notation is extended by using a functional programming language to provide a concrete version of the (traditionally unspecified) expression language which supports the process-algebra.

The existing theory is used as the basis for both a denotational semantics and an operational semantics using Plotkin-style transition systems. The two semantics are then shown to be congruent using a bridging semantics based on the synchronisation trees of Brookes, Roscoe and Walker.

A reference implementation of the semantics is outlined, along with a lexer and parser. This provides a core which can be used as the basis for future work on tools such as animators, proof assistants and refinement checkers.

Contents

1 Introduction **7**

1.1 Motivation 7

1.2 Requirements 8

1.3 Summary 9

2 Extending the syntax **10**

2.1 Starting point 10

2.2 Buffers and switches 13

2.3 Counter 13

2.4 Buffers with data 14

2.5 Buffers and sequences 15

2.6 Functions 16

2.7 Communications 17

2.8 Communication sets 18

2.9 Restricted communications 19

2.10 Vending machines 20

2.11 Renaming 21

2.12 Parameters versus renaming 23

2.13 Types and channels 23

2.14 Extensions and productions 25

2.15 Local definitions 25

2.16 Grammar 26

2.17 Summary 32

3 Denotational semantics **33**

3.1 Expression domains 33
3.2 Process domains 37
3.3 Environment 39
3.4 Booleans 40
3.5 Numbers 42
3.6 Tuples 43
3.7 Sequences 43
3.8 Sets 45
3.9 Composites 49
3.10 Binding 49
3.11 Definitions 50
3.12 Type definitions 51
3.13 Processes 53
3.14 Summary 60

4 Operational semantics **61**

4.1 Choice 63
4.2 Sequence 63
4.3 Interrupt 64
4.4 Interleave 64
4.5 Expressions 65
4.6 Communicating parallel 65
4.7 Alphabetised parallel 66
4.8 Hiding 67
4.9 Conditionals 67
4.10 Prefix 68
4.11 Renaming 68
4.12 Chaos 69
4.13 Bindings 69
4.14 Summary 70

5 Tree semantics **71**

5.1 Trees 71

5.2 Semantics 73

5.3 Connection to the operational semantics 75

5.4 Summary 78

6 Congruence **79**

6.1 Mapping to processes 79

6.2 Review 84

6.3 Congruence 88

6.4 Summary 89

7 Implementing a parser **90**

7.1 Grammar 90

7.2 Restructuring 92

7.3 Representation 93

7.4 Precedence 96

7.5 Summary 98

8 Implementing the semantics **99**

8.1 Implementation language 99

8.2 Representing a branched type 100

8.3 The syntax tree 102

8.4 Representing values 104

8.5 Environments 107

8.6 Applications 110

8.7 Booleans 112

8.8 Numbers 113

8.9 Sequences 114

8.10 Sets 116

8.11 Comprehensions 117

8.12 Processes 120
8.13 Summary 125

9 Adding extensions **127**
9.1 Timeout 127
9.2 Override 129
9.3 Boolean guards 132
9.4 Transparent operations 134
9.5 Assertions 137
9.6 Refinement 138
9.7 Summary 140

10 History **141**
10.1 Channels 141
10.2 Standard ML 142
10.3 Use of pragmas 143
10.4 Input patterns 143
10.5 Summary 143

11 Extensions **144**
11.1 Improvements 144
11.2 Additions 150
11.3 Richer models 152

12 Conclusions **154**
12.1 Summary 154
12.2 Tools 154
12.3 Applications 155
12.4 Acknowledgments 156

A Theory **158**
A.1 Existence 158

A.2 Metric 160
A.3 Fixed points 160

B Lexer and parser **163**
B.1 Ambiguity 163
B.2 Definitions and comments 164
B.3 Flex 164
B.4 Bison 168

Chapter 1

Introduction

The many dialects of CSP have been the subject of research at Oxford for well over a decade. During that time, the language outlined in [18] has formed the basis for research in timed, probabilistic, prioritised, and synchronous concurrency. However, all of these dialects have been 'blackboard' languages: they have been used for describing parallel systems when the intended audience is human.

1.1 Motivation

Experience with formal methods outside the usual academic setting suggests that techniques stand a much better chance of acceptance if they have tool support available. Those working in industry, even in research departments, believe that the problems they are given are significantly larger or require a much shorter timescale than is usual in academic research. The conclusion drawn from this belief is that application of a formal method to their particular problem will require mechanical assistance if it is to be successful.

Within an academic setting, such tools can be useful when teaching a language, since they allow students to meet the language in a more 'experimental' setting. For example, it is clear that a simple CSP animator would be useful in teaching the language.

There is also a degree of feedback; not only does the existence of tools encourage the use of a formal method, but the use of the method will in turn encourage the development of tools. This is clearly demonstrated by the growing number of tools for Z (see [36]).

Encouraging the first generation of tools for a language can be as simple as providing a common, freely available front-end. This reduces the work involved in

demonstrating a useful prototype from years to months. For example, the easy availability of the source code for a version of the Inmos occam 2 compiler made it possible for Michael Poole to retarget the compiler to generate code for the Intel family of processors, and allowed Formal Systems to produce a set of tools for the manipulation of occam 2 which were driven directly from the Inmos syntax tree. Both of these efforts would have been impractical if it was necessary to build a complete occam 2 parser, usage checker and code generator from scratch.

The fact that tools are all based on a common front-end also helps to impose some degree of standardisation on the language; for example, in some ways the Inmos compiler provides a much more practical definition of occam 2 than that given by [19]. This standardisation in turn means that tools can be used in conjunction, allowing later tools to exploit the functionality of earlier ones whenever that is useful.

We conclude that tools for CSP would help the language to achieve greater acceptance and be of use in teaching the language. Also, a practical way of encouraging such tools would to be to provide a basic but functional foundation on which tools may be built, thus eliminating the bulk of the duplicated effort between such tools and giving some degree of standardisation between the tools produced.

This thesis aims to provide a basis for the implementation of tools for CSP. Chapter 12 will list some of the tools that have built upon this work.

1.2 Requirements

Since CSP has typically been used as a blackboard language, the first stage in producing tools for CSP is to extend and modify the language to eliminate any omissions and ambiguities. Chapter 2 discusses the changes involved, illustrating the points with a series of examples. The main addition required is a language of expressions which can be used as process parameters and components of events. The resulting language is a combination of blackboard CSP and a functional programming language which has been designed to support existing CSP idioms.

Chapter 3 extends the failures-divergences model of untimed CSP to support the new language, combining the standard process model with a semantics for the expression language. A specific choice of Σ, the space of all events, connects the expression and process components.

Since the failures-divergences model describes processes using unwieldy sets and sequences of events, it is not suitable for use when implementing the language, so Chapter 4 models the process component of the semantics using a Plotkin-style transition system, while retaining the denotational model for the other parts of the

language.

Given these two separate semantics for the language, Chapter 5 defines an intermediate semantics over synchronisation trees and Chapter 6 extends the standard proof of congruence to the more detailed language used here.

Having outlined the extended language and given the supporting theory, Chapter 7 adapts the symbols used into a form which can be entered from a standard keyboard, and discusses operator precedence, working developing a grammar which can be used with standard tools like Flex and Bison.

The parser of Chapter 7 produces a syntax tree corresponding to the input program in terms of a few simple branched types. Chapter 8 discusses how such types can be represented in the chosen implementation language and then extends the same technique to model the values needed to provide an implementation of the semantics. The process components are modeled using an adaptation of the transition system of Chapter 4.

Discussion of the implementation continues in Chapter 9, as some proposed modifications are included in the language to illustrate the general method.

After a review of the differences between the language presented here and that described in earlier documents in Chapter 10, Chapter 11 outlines some possible extensions to the work including a static type-checker and an alternative approach to the operational semantics. It also covers the consequences of relaxing some of the simplifying assumptions made in the earlier chapters, such as the layering of the process model and the use of a functional rather than an imperative language to provide the mathematical toolkit.

Finally, Chapter 12 summarises the achievements of this thesis, as well as listing some of the tools and results which depend upon it. Appendices deal with some technical material from Chapters 5 and 7.

1.3 Summary

The existing forms of CSP provide a powerful mathematical notation. This thesis adapts that notation to produce a language which can be used with tools. The changes involved clarify the connection between the notation 'as used' and the underlying theory.

The language is supported using two semantic models, which are shown to be congruent, as well as a complete implementation of a parser and operational semantics. Examples demonstrate how the language may be extended to support extensions to the mathematical notation.

Chapter 2

Extending the syntax

This chapter discusses the changes which need to be made to the syntax for CSP to make it suitable for use as the input language for tools. The discussion is motivated by a series of examples. Familiarity with a dialect of CSP is assumed.

2.1 Starting point

The most accessible grammar of CSP prior to this work was probably that of [13] and was in fact a grammar for Timed CSP. However, since the timed version of the language is a strict extension of the basic untimed language, it included a grammar for the untimed version of the language.

$$
\begin{array}{rcl}
P & = & STOP \\
 & | & SKIP \\
 & | & CHAOS(A) \\
 & | & a : A \rightarrow P_a \\
 & | & P \setminus A \\
 & | & P \,;\, P \\
 & | & P \underset{a}{\triangledown} P \\
 & | & P \,|||\, P \\
 & | & P \underset{A}{\|} P \\
 & | & P \;{}_A\|_A\; P \\
 & | & P \,\Box\, P \\
 & | & P \sqcap P
\end{array}
$$

$$
\begin{array}{l}
\mid \quad \mu\, X \bullet F(X) \\
\mid \quad f[P] \\
\mid \quad f^{-1}[P] \\
\mid \quad \text{if } b \text{ then } P \text{ else } P
\end{array}
$$

where A represents a set of events, f a function between events, F a continuous function between processes, and b a boolean test.

The detail provided by this grammar has been sufficient for many papers on CSP, but it (intentionally) leaves many issues unresolved. The precise construction of the event sets, renaming functions, and boolean guards is not discussed, since this is not relevant to work on the language theory. However, they need to be defined if the language is to be used with tools.

Alphabets

The language here has evolved from that presented in [18], but its ancestry is clearly visible. The most obvious change is the explicit parameterisation of the parallel operators and the corresponding elimination of process alphabets. This change is often criticised, since it appears to require extra effort by the user: these sets must be written out, even though they were implicit in the alphabetised version of the language. In fact, the alphabets were *so* implicit in the earlier version of the language that definitions often lacked necessary information. For example, most of the proponents of the alphabetised model will quite happily write

$$P \quad = \quad STOP$$

without worrying about the ambiguity this introduces. The alphabetised model has *many* $STOP$ processes, which becomes clear in the ambiguity of

$$Q \quad = \quad (a \rightarrow STOP \,\Box\, b \rightarrow STOP) \parallel (a \rightarrow STOP)$$

where it is impossible to tell (without additional information) whether Q can perform a b action.

Even more seriously, recursions can leave alphabets under-specified: it is often desirable to allow the alphabet of

$$P \quad = \quad a \rightarrow P$$

to be larger than the default of $\{a\}$, typically by introducing clauses such as

$$\alpha P \quad = \quad \{a, b\}$$

or, less commonly, by parameterising the recursion

$$P \quad = \quad \mu\, X : \{a, b\} \bullet a \rightarrow X$$

Thus, typical usage of the alphabetised version of the language can be imprecise, and eliminating the imprecision involves either parameterising $SKIP$, $STOP$, and some recursions, or the parallel operators. Since the latter is already common in papers on the theory of the language, we also follow that approach here.

Detail

This leaves the simple syntax given above, with sixteen process productions but much of the detail left unresolved. For example, the exact structure of events (a), sets of events (A), and renaming functions (f) is rarely given explicitly; users are expected to infer reasonable usage from examples. In work exploring the theoretical aspects of the language, this approach is perfectly reasonable: how such values were obtained is not significant, merely that they exist and satisfy certain properties. If the language is to be used with tools, then this is unsatisfactory for two reasons. First, we must know how to build items for these 'holes' in the syntax. Second, these holes do not accurately reflect the language as used. For example, event prefixing typically makes use of inputs and outputs like $c?x \rightarrow P$, but the syntax above writes this $a : A \rightarrow P$ for some expressions a and A with the relation between the two forms being 'obvious.' Thus, filling in the holes will involve finding an expression language rich enough to support most common CSP idioms, and also require slight modifications to the process syntax to bring it closer to common usage.

The expression language required should support as many of the common CSP idioms as possible while satisfying several other requirements. If the extended language is to be easy to learn, it should be as simple as possible; this will also reduce the complexity of any tools which manipulate it. This requirement is in obvious conflict with directly supporting as many CSP idioms as possible. The expression language must also be implementable; while tools which manipulate the language using abstract reasoning are a possibility, there is also likely to be demand for animators and tools which explore processes mechanically. Given that CSP has many algebraic laws it is also desirable that the expression language be amenable to algebraic manipulation.

The examples that follow will examine some simple CSP examples and consider the impact each of these has on the expression language and the details of the process syntax.

2.2 Buffers and switches

The first example is a two-state process which arises in many different ways. For example, it corresponds to a switch

$$SWITCH \quad = \quad on \rightarrow off \rightarrow SWITCH$$

but it also corresponds to a one-place buffer when the precise details of the data have been abstracted

$$\begin{array}{rcl} EMPTY & = & in \rightarrow FULL \\ FULL & = & out \rightarrow EMPTY \end{array}$$

The first observation to be made is that recursive definitions are commonly built by using named terms rather than the explicit $\mu X \bullet F(X)$ notation; the latter is typically used only to avoid introducing a named expression where it is awkward to do so.

The second observation is that the events in the above fragments were never declared. Experience with languages (like BASIC), which allow use of identifiers without declaration, suggests that this leads to the dangerous situation where trivial keyboard errors can introduce subtle problems which are difficult to locate. To provide a measure of safety (and to simplify the semantics and implementation) we add

channel in, out, on, off

to indicate that the identifiers may be used as events (the channel keyword has been adapted from the truncated `CHAN` keyword of occam.)

2.3 Counter

One of the examples used in [18] is the simple counter which is allowed to move up and down, but is allowed to move around only when it is at ground-level. This may be written as

channel $up, down, around$

$$\begin{array}{rcl} COUNTER & = & COUNT(0) \\ COUNT(n) & = & \left\{ \begin{array}{l} \text{if } n = 0 \text{ then} \\ \quad up \rightarrow COUNT(n+1) \,\Box\, around \rightarrow COUNT(n) \\ \text{else} \\ \quad up \rightarrow COUNT(n+1) \,\Box\, down \rightarrow COUNT(n-1) \end{array} \right. \end{array}$$

The process arguments, which are traditionally written using subscripts, are here written in parentheses following the process name. The subscript convention causes problems with the ASCII representation of the syntax discussed later and also becomes difficult to read when many subscripts are involved.

The example here uses natural numbers as a process argument, but of course integers may be required in general, along with a much richer space of values.

The version of the counter defined in [18] uses an infinite set of equations of the form

$$\begin{aligned} CT_0 &= (up \rightarrow CT_1 \,|\, around \rightarrow CT_0) \\ CT_{n+1} &= (up \rightarrow CT_{n+2} \,|\, down \rightarrow CT_n) \end{aligned}$$

This effectively uses '$n+1$' pattern-matching to express the conditional. While such pattern-matching definitions can be useful, they are not strictly necessary and we omit them for simplicity. Section 11.1 discusses the work involved in adding pattern-matching to the language.

2.4 Buffers with data

Adding more detail to the rather abstract buffer given earlier involves describing the data held by the buffer and associating this data with the channels along which it is communicated. For example, a buffer holding a single integer is given by

$$\begin{aligned} &\text{channel } in, out : Int \\ &INTBUFF = in?x \rightarrow out!x \rightarrow INTBUFF \end{aligned}$$

The useful notation of ? and ! is used for communications, rather than the abstract $a : A$ form. There is a minor problem with the fragment $a : A \rightarrow P_a$ in that the precise mechanism by which the body, P_a, depends on the event chosen is unclear. Using ? and ! fields makes this obvious: identifiers are bound by inputs and their scope is the body of the communication. This was used in the above fragment without mention; the x that is output is that which was previously input.

The input values are *bound* to the identifiers and the binding disappears outside the body of the communication. Thus

$$xNotBound = (in?x \rightarrow SKIP)\,;(out!x \rightarrow SKIP)$$

is *not* equivalent to the above buffer; supporting this usage would require some notion of 'state' to be transferred across sequential composition. See [16] for some of the complications that this introduces.

Note that there are some restrictions on the types which may be used in the construction of a channel. Intuitively, if synchronisation is to be performed on events, there must be an effective definition of equality for the types involved. Since equality between functions cannot always be decided, it is not practical to permit functions as part of events. Although such communications are appealing, they conflict with the way in which CSP manufactures communications from a more general synchronisation mechanism. Similarly, processes may not form part of channels. (The restriction on processes is also necessary to allow the construction of the standard semantic models. The huge theoretical difficulties caused by allowing communication of processes are outside the scope of this thesis.)

2.5 Buffers and sequences

Multi-place buffers are traditionally coded using a sequence to hold the current contents of the buffer

$$\begin{array}{l} \text{channel } in, out : Int \\ BUFF(s) \quad = \quad \left\{ \begin{array}{l} \text{if } null(s) \text{ then} \\ \quad in?x \rightarrow BUFF(\langle x \rangle) \\ \text{else} \\ \quad in?x \rightarrow BUFF(s^\frown\langle x \rangle) \,\Box\, out!head(s) \rightarrow BUFF(tail(s)) \end{array} \right. \end{array}$$

The *head* and *tail* operations are used to decompose sequences, rather than Hoare's s_0 and s'. As mentioned earlier, the pattern-matching form of the definition is avoided, so a conditional expression is used, rather than using separate definitions for $BUFF(\langle\rangle)$ and $BUFF(\langle x \rangle^\frown s)$. See subsection 11.1 for a detailed discussion of the work involved in supporting pattern-matching.

Sequences play a similar role in CSP to the lists of functional programming languages, so we adapt the list-comprehensions of those languages for use with sequences along with the restriction that elements of a sequence should all have the same type. Such comprehensions appear to be easier to learn than the Lisp equivalents of maps and filters. The most basic form of a comprehension is a simple enumeration of values, as in

$$\begin{array}{lcl} s & = & \langle 2, 4, 6, 8, 10 \rangle \\ t & = & \langle 1, 2, 3, 4, 5 \rangle \end{array}$$

with range expressions to simplify the latter to

$$t \quad = \quad \langle 1..5 \rangle$$

and a similar, infinite range

$$u = \langle 3.. \rangle$$

Generators allow the construction of more complex sequences. An identifier is bound to each value in turn from a sequence and the results combined. For example, s may also be written as

$$s = \langle x * 2 \mid x \leftarrow \langle 1..5 \rangle \rangle$$

Filters allow us to be selective about which values from a generator are passed through. For example, s may also be written as

$$s = \langle x \mid x \leftarrow \langle 1..10 \rangle, x \% 2 = 0 \rangle$$

as this selects the numbers between 1 and 10 which are even. (We use % for the integer remainder operation.)

Generators are unrolled so that the right-most generator 'runs fastest', with the current value of generators to the left of the current one available. For example, if

$$u = \langle (x, y) \mid x \leftarrow \langle 1..3 \rangle, y \leftarrow \langle x..3 \rangle \rangle$$

then this is the same as

$$u = \langle (1,1), (1,2), (1,3), (2,2), (2,3), (3,3) \rangle$$

Since traditional CSP also makes use of sets, we add these to the language as well and allow them to use the same comprehension mechanism, although of course here the generators take their values from other set expressions. There is also a function (set) from sequences to sets.

2.6 Functions

As well as processes with parameters, we often need to define functions; for example we might model a process which squares its inputs using

$$\begin{aligned} square(y) &= y * y \\ SQUARE &= in?x \rightarrow out!square(x) \rightarrow SQUARE \end{aligned}$$

and in fact, using function arguments we can generalise this to

$$OPBUFF(f) \quad = \quad in?x \rightarrow out!f(x) \rightarrow OPBUFF(f)$$

and then write

$$SQUARE \quad = \quad OPBUFF(square)$$

Together with sets and sequences, functions provide the core of the expression language. They provide a large part of the expressive power of functional languages such as Orwell/Miranda and Gofer/Haskell (see [21] and [6]).

2.7 Communications

Although the examples in 2.4 and 2.6 use simple inputs and outputs, CSP permits more complex communications. Rather than dividing communications into inputs and outputs, CSP implements both of these using a general multi-way synchronisation mechanism. For example, consider a synchronisation on channel c

$$\begin{array}{l} \text{channel } c : Int \\ P \quad = \quad \left\{ \begin{array}{c} (c.1 \rightarrow STOP \,\Box\, c.2 \rightarrow STOP) \\ \underset{c}{\parallel} \\ (c.2 \rightarrow STOP \,\Box\, c.3 \rightarrow STOP) \end{array} \right. \end{array}$$

where P performs $c.2$ since that is the only event on which all of the processes involved can agree. If the intersection contains multiple events, then the choice is propagated outwards

$$\left. \begin{array}{c} (\Box\, i : \{1,2,3\} \bullet c.i \rightarrow STOP) \\ \underset{c}{\parallel} \\ (\Box\, i : \{2,3,4\} \bullet c.i \rightarrow STOP) \end{array} \right\} \quad = \quad \Box\, i : \{2,3\} \bullet c.i \rightarrow STOP$$

If we restrict this mechanism to two processes, one of which (the 'sender') is offering a single value, then the branch taken by the 'receiver' is under the control of the sender.

$$c!7 \rightarrow STOP \underset{c}{\parallel} c?x \rightarrow P(x) \quad = \quad c.7 \rightarrow (STOP \underset{c}{\parallel} P(7))$$

Thus, 'outputs' are communications which offer one value and 'inputs' are communications which offer a selection and bind an identifier to the value communicated.

In CSP, unlike occam, a single synchronisation can communicate values in both directions. For example

$$\left.\begin{array}{l}\text{channel } d : Int.Int \\ (\Box\, i : Int \bullet d.i.5 \rightarrow P(i)) \\ \qquad \underset{d}{\|} \\ (\Box\, j : Int \bullet d.2.j \rightarrow Q(j))\end{array}\right\} \quad = \quad d.5.2 \rightarrow (P(2) \underset{d}{\|} Q(5))$$

Allowing inputs and outputs to be mixed in a single term, this can be written more concisely as

$$d?i!5 \rightarrow P(i) \underset{d}{\|} d!2?j \rightarrow Q(j) \quad = \quad d.5.2 \rightarrow (P(2) \underset{d}{\|} Q(5))$$

This generalises to multi-way synchronisation and communication, for example

$$\left.\begin{array}{l}\text{channel } e : Int.Int.Int \\ e!1?i?j \rightarrow P(i,j) \\ \qquad \underset{e}{\|} \\ e?i!2?j \rightarrow Q(i,j) \\ \qquad \underset{e}{\|} \\ e?i?j!3 \rightarrow R(i,j)\end{array}\right\} \quad = \quad e.1.2.3 \rightarrow (P(2,3) \underset{e}{\|} Q(1,3) \underset{e}{\|} R(1,2))$$

2.8 Communication sets

In the examples above, the processes shared a single channel and this was written simply by placing the channel name underneath the parallel operator, as in $P \underset{c}{\|} Q$. This is a standard convention when using CSP as a blackboard language and is extended to cover multiple channels by placing the channels in braces, as in $P \underset{\{c,d,e\}}{\|} Q$.

However, in general the sets controlling such synchronisations are not so regular; they are arbitrary sets of events as in

$$\begin{array}{l}\text{channel } c : Int \\ P \quad = \quad Q \underset{\{c.0, c.2, c.5\}}{\|} R\end{array}$$

While it would be natural to define $\{c, d\}$ as denoting the set of events associated with the channels c and d, this is awkward; the expression $\{c, d\}$ also has a meaning

as the set of channels as in

$$\begin{array}{l}\text{channel } c, d : Int \\ P = \Box\, x : \{c, d\} \bullet x?y \rightarrow STOP\end{array}$$

However, the use of complete channels to generate communication sets is so common that it is essential to provide a concise notation for generating sets of events from channels: it is totally unacceptable to insist that $P = Q \underset{c}{\parallel} R$ be written as

$$\begin{array}{l}\text{channel } c : Int \\ P = Q \underset{\{c.n \mid n \in Int\}}{\parallel} R\end{array}$$

Since it is necessary to distinguish the set of events corresponding to some channels from the set of those channels themselves, we introduce a separate notation for the first of these; it is written as $\{|c, d|\}$. The form $\{c, d\}$ is used to denote the set of channels. The additional vertical bars may be read as marking the set as being associated with communication, or alternatively as 'closing' the set under the event construction operation; the latter interpretation will be discussed in more detail in Section 2.14.

2.9 Restricted communications

Using a mixture of ! and ?, the shapes of the sets of events which may be offered on a channel with multiple events are limited: the decision at each stage is either an input (all possible values) or an output (a single value). In practice, this is far too restrictive. For example, given

$$\begin{array}{l}\text{channel } c : \{0..9\} \\ evens \;=\; \{2 * n \mid n \leftarrow \{0..4\}\}\end{array}$$

we might want to offer the set of even events, without using the cumbersome

$$P \;=\; \Box\, i : evens \bullet c.i \rightarrow STOP$$

To support this, there is a restricted form of the ? operation which limits the set of values offered. For example, the above fragment can be written as

$$P \;=\; c?i : evens \rightarrow STOP$$

This addition allows sets of different 'widths' to be constructed, but they are still rectangular. This restriction is eliminated if we allow previously bound input variables to be used in later inputs and outputs. For example, we can input any pair of integers which are in ascending order using

$$\begin{array}{l} \text{channel } d : Int.Int \\ P \quad = \quad d?x?y : \{x..\} \rightarrow SKIP \end{array}$$

2.10 Vending machines

Integers, sets, and sequences are common and useful, but it is often helpful to be able to define types. For example, consider a vending machine offering a variety of canned drinks, where we refer to the drinks by name (with apologies to LucasFilm)

$$\text{datatype } Drink = Grog \mid DietGrog \mid CherryGrog$$

A faulty vending machine may then be described as

$$\begin{array}{l} \text{channel } can : Drink \\ \text{channel } coin \\ \quad VM \quad = \quad coin \rightarrow PAID \\ PAID \quad = \quad \left\{ \begin{array}{l} can?x : Drink \setminus \{Grog\} \rightarrow VM \\ \Box \\ can.Grog \rightarrow PAID \end{array} \right. \end{array}$$

where the machine fails to notice that *Grog* has been dispensed; customers may take as many of these as they wish.

The machine owners eventually notice a problem, which the users seemed reluctant to report, and rush out a quick patch which attempts to enforce the requirement that all drinks must be paid for.

$$PATCH \quad = \quad coin \rightarrow can?x \rightarrow PATCH$$

The patch is applied to give a 'fixed' vending machine.

$$VM2 \quad = \quad VM \underset{\{|coin, can|\}}{\parallel} PATCH$$

where $\{|coin, can|\}$ denotes the set of all events on those two channels.

Customers are extremely unhappy with the patched machines; not only do they refuse to let them have drinks for free, but as soon as they try to exploit the old

bug, the vending machines deadlock, and they become unable to obtain drinks legitimately.

The vending machine suppliers, reasoning that the only thing that can be done to repair a deadlocked machine is to reset it, fit the machine with a reset line.

channel *reset*
$VM3 = VM2 \triangle reset \rightarrow VM3$

The (new) interrupt operator used here replaces that of [13] and is closest to that of [18]. The process to the left of the operator runs until an initial action of the process on the right occurs, at which point control is transferred to that process. In the case given here, the *reset* event takes the vending machine back to its initial configuration.

Given this reset line, the final patch simply resets the machine after every can of *Grog* is dispensed.

$$\begin{aligned} PATCH2 &= can.Grog \rightarrow reset \rightarrow PATCH2 \\ VM4 &= (VM3 \underset{\{|reset, can.Grog|\}}{\parallel} PATCH2) \setminus \{|reset|\} \end{aligned}$$

Of course, the fully-fixed vending machine is now equivalent to the machine where the bug has been fixed properly, so that $VM4 = VM5$ where

$$VM5 = coin \rightarrow can?x \rightarrow VM5$$

2.11 Renaming

CSP as presented in [18] uses a simple form of renaming; given a process P and an injective function f on the set of events, we are allowed to form $f(P)$ and $f^{-1}(P)$. Later work generalises this and allows the restriction to injective functions to be relaxed. As Davies notes in [13], this allows us to express many-to-one and one-to-many relations using forward (f) and backward (f^{-1}) functional-renaming. For example, given

$$\begin{aligned} P &= a \rightarrow STOP \,\Box\, b \rightarrow c \rightarrow STOP \\ f(a) &= x \\ f(b) &= y \\ f(c) &= y \end{aligned}$$

then we have

$$f(P) = x \rightarrow STOP \,\Box\, y \rightarrow y \rightarrow STOP$$

identifying the events b and c, while

$$\begin{array}{rcl} f^{-1}(f(P)) & = & a \rightarrow STOP \\ & \Box & b \rightarrow (b \rightarrow STOP \,\Box\, c \rightarrow STOP) \\ & \Box & c \rightarrow (b \rightarrow STOP \,\Box\, c \rightarrow STOP) \end{array}$$

so that every y presented is replaced by a choice between b and c. (If the standard semantic models are used, it is necessary to insist that for every event e, the set mapped to e is finite in order to preserve finite non-determinism.)

Any renaming which is many-to-many has to be constructed from a combination of forward and backward renaming; this is often inconvenient since it may be necessary to invent intermediate events which connect the two halves. Using functions to describe renaming has other problems; defining the functions requires a powerful pattern-matching mechanism; the definitions are often bulky and break the natural flow of a process definition. Finally, the convention that unmentioned events must be left unchanged would require special treatment for renaming functions. For example, in the above example, given

$$\text{channel } a, b, c, d, x, y, z$$

it would be considered 'obvious' that

$$\begin{array}{rcl} f(d) & = & d \\ f(z) & = & z \end{array}$$

We avoid all of these problems by replacing this notation with another. Since renamings are in general relations between events, the notation simply creates the relation as part of the operation. We use an 'assignment' arrow ($\leftarrow$) to denote 'becomes', and we can then write $f(P)$ as Q where

$$Q \;=\; P \,[\![a \leftarrow x, b \leftarrow y, c \leftarrow y]\!]$$

with the assumption that events which are not mentioned on the left of an arrow are to be left unchanged. Similarly $f^{-1}(Q)$ can be written as R where

$$R \;=\; Q \,[\![x \leftarrow a, x \leftarrow x, y \leftarrow b, y \leftarrow c, y \leftarrow y]\!]$$

This notation extends naturally to a comprehension form for more complex renamings

$$\begin{array}{l} \text{channel } c : \{0..7\} \\ P \;=\; c!2 \rightarrow c!4 \rightarrow STOP \\ Q \;=\; P \,[\![c.n \leftarrow c.7-n \mid n \leftarrow \{0..7\}]\!] \end{array}$$

which gives

$$Q \;=\; c!5 \rightarrow c!3 \rightarrow STOP$$

2.12 Parameters versus renaming

The terms in the renaming do not have to be events; if channels are used then the renaming extends pointwise to all the channel members. For example, we can chain processes together, as in

$$\begin{array}{l}\text{channel } in, mid, out : Int \\ P \quad = \quad (Q \,[\![out \leftarrow mid]\!] \underset{\{|mid|\}}{\parallel} R \,[\![in \leftarrow mid]\!]) \setminus \{|mid|\}\end{array}$$

This approach to 'chaining' processes is used in [18] where it is denoted by a special operator, $\gg$. Unfortunately, it fails to work well when channels must be declared. In the example above, the same channel declarations apply to Q and R which in turn implies that the input channels of these processes must have the same type, as must the output channels. The problem is caused by the way in which channels used by Q and R are simply captured from the environment; if they were explicit parameters, then this type restriction could have been avoided. For example, if we have

$$\begin{array}{lcl} Q(in, out) & = & in?x \rightarrow out!\langle x \rangle \rightarrow Q(in, out) \\ R(in, out) & = & in?x \rightarrow out!\{x\} \rightarrow R(in, out) \end{array}$$

then we can write

$$\begin{array}{l}\text{channel } in : Int \\ \text{channel } mid : Seq(Int) \\ \text{channel } out : Set(Seq(Int)) \\ P(in, out) \quad = \quad (Q(in, mid) \parallel R(mid, out)) \setminus \{|mid|\}\end{array}$$

As this suggests, it is better to make channels parameters to processes, rather than renaming the channels later.

2.13 Types and channels

Let us imagine a new, enhanced vending machine which in addition to the drinks dispensed by the earlier versions can also provide chocolate. Then, as before, we have

$$\text{datatype } Drink = Grog \mid DietGrog \mid CherryGrog$$

and the machines dispenses either a chocolate, or a can of drink

$$\begin{array}{l}\text{datatype } Product = Chocolate \mid Can.Drink \\ \text{channel } product : Product\end{array}$$

and we can build the machine simply using

$$\begin{array}{l}\text{channel } clunk \\ VM6 \quad = \quad coin \rightarrow clunk \rightarrow VM6 \\ VM7 \quad = \quad VM6 \, [\![clunk \leftarrow product.x \mid x \leftarrow Product]\!]\end{array}$$

We can then construct a customer who accepts any product,

$$CUST1 \quad = \quad coin \rightarrow product?x \rightarrow CUST1$$

a customer who will accept only a drink,

$$CUST2 \quad = \quad coin \rightarrow product.Can?x \rightarrow CUST2$$

or a customer who insists on chocolate

$$CUST3 \quad = \quad coin \rightarrow product.Chocolate \rightarrow CUST3$$

In fact, we can treat the branches of *product* as if they were channels in their own right. For example, we can count the number of chocolates and drinks dispensed by using

$$MONITOR \quad = \quad DRINKS(0) \,|||\, CHOCS(0)$$

where

$$\begin{array}{rcl} DRINKS(n) & = & product.Can?x \rightarrow DRINKS(n+1) \\ CHOCS(n) & = & product.Chocolate \rightarrow CHOCS(n+1) \end{array}$$

Then the 'monitored' vending machine is

$$VM8 \quad = \quad VM7 \underset{\{|product|\}}{\parallel} MONITOR$$

Of course, the monitor has a design flaw in that the information it records can never be extracted.

2.14 Extensions and productions

The special form of set $\{|c, d|\}$ was introduced for the purposes of concisely describing the set of all events associated with the two channels c and d. In fact this is a specific instance of a much more general mechanism. These special sets use two related operations, called *extensions* and *productions*, to calculate the required value.

The tags associated with datatype definitions, (*Can* and *Chocolate* in the previous section) are closely related to the identifiers which are used to denote channels. The channel definitions simply add a new branch to the built-in 'Event' datatype, which is special only in that it may be incrementally extended in this way.

For any partially completed datatype value, the function *extensions* calculates from the type definition the set of values whose addition would complete the value. Thus, in the above example $extensions(Can) = Drink$. Similarly, if we declare

$$\text{channel } multiple : \{1..4\}.\{1..3\}.Drink$$

then $extensions(multiple.1) = \{n.d \mid n \leftarrow \{1..3\}, d \leftarrow Drink\}$.

The *productions* function returns the set of values which *are* the completions of that type, so that

$$productions(x) \quad = \quad \{x.y \mid y \leftarrow extensions(x)\}$$

(Note that for values which are already complete, this definition requires that *extensions* return the set consisting of a single value which is a right-identity for the dot operation. Such a value exists, but has no direct representation in the syntax.)

Given these definitions, it should be clear that $\{|c|\}$ may be regarded as syntactic-sugar for $productions(c)$ since both denote the set of events associated with channel c, and also that the bars inside the set can be considered as closing the set to produce all possible productions. Thus the sets may also be used with datatypes, so that

$$\{|Can|\} \quad = \quad \{Can.Grog, Can.DietGrog, Can.CherryGrog\}$$

2.15 Local definitions

There are often functions associated with a single process definition and it is convenient to avoid the need to find unique names for these functions by localising

the definitions. Blackboard CSP provides no direct mechanism for this, relying on obvious 'locality' of definitions to associate them with processes and resolve conflicting use of identifiers. We introduce a 'let' clause for this purpose. For example, the process which outputs the prime numbers can be written (inefficiently) as

$$
\begin{array}{l}
\text{channel } prime : Int \\
FILT(c,p,n) \;=\; \left\{ \begin{array}{l} \text{let} \\ \quad P(n) = \text{if } p(n) \text{ then } c!n \rightarrow P(n{+}1) \text{ else } P(n{+}1) \\ \text{within } P(n) \end{array} \right. \\
PRIMES \;=\; \left\{ \begin{array}{l} \text{let} \\ \quad factors(n) = \langle x \mid x \leftarrow \langle 2..n{-}1 \rangle, n\%x = 0 \rangle \\ \quad is_prime(n) = null(factors(n)) \\ \text{within } FILT(prime, is_prime, 2) \end{array} \right.
\end{array}
$$

Here the definition of *PRIMES* uses a 'let' clause to conceal the auxiliary functions, while the definition of *FILT* uses one to capture the channel and predicate parameters. As a result the recursive definition of P uses only one parameter.

Given the functional flavour to the expression language, we also add support for the nameless function provided by lambda terms.

2.16 Grammar

We now outline a grammar which contains all of the examples presented above. The grammar will be used for the purposes of giving various semantics to the language in the following chapters, but is not suitable for producing a parser. We will return to the grammar and discuss the changes necessary for that in Chapter 7.

Processes

The revised grammar for process terms is

$$
\begin{array}{rcl}
Proc & = & STOP \\
& | & SKIP \\
& | & CHAOS(A) \\
& | & Chan\ Fields \rightarrow Proc \\
& | & Proc \setminus A \\
& | & Proc\ ;\ Proc \\
& | & Proc \triangle Proc
\end{array}
$$

$$
\begin{array}{rcl}
 & | & Proc \,|||\, Proc \\
 & | & Proc \underset{A}{\|} Proc \\
 & | & Proc \;{}_A\|_A\; Proc \\
 & | & Proc \,\Box\, Proc \\
 & | & Proc \,\sqcap\, Proc \\
 & | & Proc \,[\![Rename]\!] \\
 & | & \text{if } Bool \text{ then } Proc \text{ else } Proc \\
 & | & \text{let } Defns \text{ within } Proc \\
 & | & Common
\end{array}
$$

where we use *Common* to indicate a named term. This also includes function applications since these correspond to parameterised named terms.

$$
\begin{array}{rcl}
Common & = & Name \\
 & | & Expr(Exprs) \\
Exprs & = & Expr \\
 & | & Expr, Exprs
\end{array}
$$

There are five differences from the grammar given in Section 2.1.

1. Explicit recursion has been replaced by named terms.
2. Input and output have been made explicit.
3. The interrupt operator has been revised.
4. Renaming is relational rather than functional.
5. Local definitions have been added.

These were discussed in sections 2.2, 2.4, 2.10, 2.11 and 2.15 respectively.

Although the above grammar uses *Bool*, *Chan*, and *A* to indicate boolean, channel and set expressions, the grammar of expressions *Expr* given in the next section does not distinguish these. Hence we can take

$$
\begin{array}{rcl}
Bool & = & Expr \\
Chan & = & Expr \\
A & = & Expr
\end{array}
$$

The *Fields* production indicates a (possibly empty) sequences of inputs and outputs, so

$$\begin{array}{rcl} \mathit{Fields} & = & \epsilon \\ & | & \mathit{Field}\ \mathit{Fields} \end{array}$$

where

$$\begin{array}{rcl} \mathit{Field} & = & ?\mathit{Expr} \\ & | & ?\mathit{Expr} : \mathit{Expr} \\ & | & !\mathit{Expr} \end{array}$$

We defer defining *Rename* until later because it requires the comprehension mechanism used in expressions.

Expressions

Expressions include basic productions for common terms and conditionals

$$\begin{array}{rcl} \mathit{Expr} & = & \mathit{Common} \\ & | & \text{if } \mathit{Bool} \text{ then } \mathit{Expr} \text{ else } \mathit{Expr} \end{array}$$

We provide basic arithmetic operations

$$\begin{array}{rcl} \mathit{Expr} & = & \mathit{Number} \\ & | & \mathit{Expr} + \mathit{Expr} \\ & | & \mathit{Expr} - \mathit{Expr} \\ & | & \mathit{Expr} * \mathit{Expr} \\ & | & \mathit{Expr} / \mathit{Expr} \\ & | & \mathit{Expr}\ \% \ \mathit{Expr} \\ & | & -\mathit{Expr} \end{array}$$

Similarly, we have the logical operations

$$\begin{array}{rcl} \mathit{Expr} & = & \text{true} \\ & | & \text{false} \\ & | & \mathit{Expr} \wedge \mathit{Expr} \\ & | & \mathit{Expr} \vee \mathit{Expr} \\ & | & \neg\ \mathit{Expr} \end{array}$$

and the comparison operations

$$\begin{array}{rcl} \mathit{Expr} & = & \mathit{Expr} = \mathit{Expr} \\ & | & \mathit{Expr} \neq \mathit{Expr} \\ & | & \mathit{Expr} < \mathit{Expr} \\ & | & \mathit{Expr} \leqslant \mathit{Expr} \\ & | & \mathit{Expr} > \mathit{Expr} \\ & | & \mathit{Expr} \geqslant \mathit{Expr} \end{array}$$

For tuples, lambda terms, and dotted expressions we have

$$\begin{array}{rcl} \mathit{Expr} & = & (\mathit{Exprs}) \\ & | & \lambda\, \mathit{Exprs}.\mathit{Expr} \\ & | & \lambda\, \mathit{Exprs}.\mathit{Proc} \\ & | & \mathit{Expr}.\mathit{Expr} \end{array}$$

For simple sets we take

$$\mathit{Expr} \;=\; \{\mathit{Targ}\}$$

where the target *Targ* is either empty, a non-empty list of values, a closed range, or an open range.

$$\begin{array}{rcl} \mathit{Targ} & = & \epsilon \\ & | & \mathit{Exprs} \\ & | & \mathit{Expr}..\mathit{Expr} \\ & | & \mathit{Expr}.. \end{array}$$

More complex set comprehensions, like $\{n \mid n \leftarrow \{0..5\}, n\%2 = 0\}$, add a list of generators

$$\mathit{Expr} \;=\; \{\mathit{Targ} \mid \mathit{Gens}\}$$

where each generator is either a boolean filter, or a map

$$\begin{array}{rcl} \mathit{Gens} & = & \mathit{Gen} \\ & | & \mathit{Gen}, \mathit{Gens} \\ \mathit{Gen} & = & \mathit{Bool} \\ & | & \mathit{Expr} \leftarrow \mathit{Expr} \end{array}$$

Sequences and communication sets are built in the same way

$$\begin{array}{rcl} Expr & = & \langle Targ \rangle \\ & | & \langle Targ \mid Gens \rangle \\ & | & \{| Targ |\} \\ & | & \{| Targ \mid Gens |\} \end{array}$$

We also have operations on sets and sequences

$$\begin{array}{rcl} Expr & = & Expr \cup Expr \\ & | & Expr \cap Expr \\ & | & Expr \setminus Expr \\ & | & \bigcup Expr \\ & | & \bigcap Expr \\ & | & \mathrm{card}(Expr) \\ & | & Expr \in Expr \\ & | & Expr^\frown Expr \\ & | & \mathrm{concat}(Expr) \\ & | & \#Expr \\ & | & Expr \text{ in } Expr \end{array}$$

The relations used in the renaming operator (which has replaced the functional and inverse-functional renamings) also use GS to allow the construction of complex renamings, but they replace the target with a non-empty list of maplets.

$$\begin{array}{rcl} Rename & = & Maps \\ & | & Maps \mid Gens \\ Maps & = & Map \\ & | & Map, Maps \\ Map & = & Expr \leftarrow Expr \end{array}$$

Replicated process operators

When describing large networks of processes, it is often useful to have replicated forms of the process operators. Their semantics can be defined directly in terms of the binary forms of the operators.

$$Proc = |||\, Gens \bullet Proc$$

$$
\begin{array}{rcl}
 & | & \|_{Expr}\ Gens \bullet Proc \\
 & | & \|\ Gens \bullet (Expr, Proc) \\
 & | & \Box\ Gens \bullet Proc \\
 & | & \Pi\ Gens \bullet Proc \\
 & | & ;\ Gens \bullet Proc
\end{array}
$$

These are used in preference to operators such as $|||\{Proc \mid Gens\}$ since those operators require forming sets of processes. Sets of processes are not compatible with the standard semantic model. As, in general, equality on processes is not effectively computable.

Note that the replicated form of sequential composition is dependent upon the order of its components, so the generators in these case are required to describe a sequence rather than a set.

Definitions

Local definitions blocks, and entire scripts, are built from a number of definitions. Those can be be simple definitions, channel definitions, or datatype definitions.

$$
\begin{array}{rcl}
Defns & = & Defn \\
 & | & Defn\ Defns \\
Defn & = & Expr = Expr \\
 & | & Proc = Proc \\
 & | & \text{channel } Names : Type \\
 & | & \text{channel } Names \\
 & | & \text{datatype } Name = VarType
\end{array}
$$

where the supporting terms are

$$
\begin{array}{rcl}
Names & = & Name \\
 & | & Name, Names \\
Type & = & Type \\
 & | & Type.Type \\
 & | & (Types) \\
 & | & Expr \\
 & | & Name \\
 & | & Name(Types)
\end{array}
$$

$$
\begin{array}{rcl}
VarType & = & Branch \\
& | & Branch \mid VarType \\
Branch & = & Name \\
& | & Name.Type \\
Types & = & Type \\
& | & Type, Types
\end{array}
$$

2.17 Summary

Starting from the process syntax of [13], we have filled in the details of the expression components using a functional programming language. That language has been designed to support as many of the idioms of CSP as possible. The features of the language being motivated by examples of their use, and a grammar has been constructed for the new, enriched language.

The next chapter gives a denotational semantics for this language.

Chapter 3

Denotational semantics

The failures-divergences model is the standard denotational semantics for CSP. This chapter extends that model to cope with the expression language of Chapter 2.

In the same way that the original syntax for CSP presented in Section 2.1 ignores the expressions used in subscripts and prefixing, the standard denotational models for CSP take the event set Σ as given. The failures-divergences model is built using

$$\mathcal{N} \subset \mathbb{P}(\Sigma^* \times \mathbb{P}(\Sigma)) \times \mathbb{P}(\Sigma^*)$$

The fact that $\mathcal{N}$ is defined as a set using Σ implies that it is not possible to make Σ depend upon $\mathcal{N}$ without deviating significantly from this approach; this is related to the fact that the standard syntax does not permit communication of processes.

3.1 Expression domains

If the standard model is to be used, we must provide a concrete model for Σ which connects the supporting expression language to the process language. Events are typically built from expression values using dots, as are composite values in the expression language, but it is useful to impose additional restrictions on events.

Finite values

Firstly, to simplify any implementation, we have a special *event* type whose elements may be communicated; these are built using dot from an anchor which was declared to be a *channel*. This allows us to separate out events (which may be communicated) from arbitrary values.

If the communication is intended to be implementable, then there are some values in the expression domain which it is not reasonable to transfer between processes. Communications in physical systems (like those of occam) are typically built in terms of a simple transfer of a single integer value. By chaining these together it is possible to support richer transfers, but these expand into a finite number of small transactions. From an operational viewpoint it therefore makes sense to restrict communication values to be 'finite'. This is in contrast to general expression values where the basic design calls for a lazy, functional approach.

From a more theoretical point of view, we require a notion of equality on the events in Σ when calculating synchronisations between processes. Mathematically this is not a problem since we are given this equality for free. However, an implementation of the semantics will require an implementation of this equality. Because any implementation of the semantics is most likely to diverge itself, attempting to transmit an infinite value should cause the process to diverge in the semantics,

We must therefore limit Σ to literals and finite structures built from these using sets, sequences, tuples, and dots. Similarly, we may wish to implement all sets in ways that require evaluation of their components (for example lists without duplicates, or hash tables), so we must build all sets out of such finite values.

If we let F denote the set of such finite values, then we require that F contains

- N the space of numbers.
- B the space of booleans.
- L the space of labels (for channels and types.)

Note that L is assumed to contain a distinguished element $\checkmark$ which is used to denote process termination. It is used internally by the semantics, but cannot be generated directly by any user syntax; this allows all the semantic models to assume that $\checkmark$ is final (no process may perform any actions after performing $\checkmark$.)

F must also contain finite sets, sequences, tuples, and dotted combinations of these. However, dotted combinations are *flat*, in that they have no further direct dotted components in them. Let $\langle X \rangle_f$ denote the set of all finite sequences over X, X^* denote the set of tuples over X, $\mathbb{F}(X)$ denote the finite power-set over X, and $X^\bullet$ denote all finite lists built using dot from values in X, including the singletons and the empty production. Then X is embedded within $X^\bullet$ in a natural way and we may take

$$\begin{aligned} F_0 &= (N \cup B \cup L)^\bullet \\ F_{n+1} &= (N \cup B \cup L \cup \mathbb{F}(F_n) \cup \langle F_n \rangle_f \cup {F_n}^*)^\bullet \\ F &= \bigcup_{n \in \mathbb{N}} F_n \end{aligned}$$

We may then identify Σ with the subset of F whose elements consist of a number of values in F joined using the dot operation and 'anchored' to a label from L.

Building the domain

Given the set F, we now consider E, the space of all expression values, excluding processes. We may wish to exploit the 'infinite' values permitted by lazy-evaluation, so the restriction to finite values, used in F, is too strong to be applied here. For E we thus relax the sequence construction operation to the domain constructor defined by

$$\langle X \rangle \quad = \quad \{\langle\rangle\} + X \times \langle X \rangle$$

The tuple and dot operations used when building F can similarly be replaced with domain constructors. However, replacing the finite-powerset operation is more problematic. The full powerset operation is not a domain constructor, and the standard powerdomain constructions require that the sets must be closed (either upward, downward, or convexly) which is not suitable for the semantics given later.

However, it is insufficient to allow only finite sets; for example many synchronisation sets are naturally infinite. If we simply use $\mathbb{P}(F)$ as a (non-recursive) component of E, then there are still problems. For example, if we consider

$$\begin{array}{rcl} f(n) & = & \text{if } n \leqslant 1 \text{ then } n \text{ else } f(n) \\ s & = & \{f(n) \mid n \leftarrow \{1..\}\} \end{array}$$

then, theoretically, we can see that $s = \{1, \bot\}$ but it is unclear whether an implementation can reduce $1 \in s$ to a non-bottom value. Any implementation is likely to place *some* enumeration order on the set, and this will control whether $\bot$ or 1 is produced first. If we ignore this ordering, the best we can do is to insist that the above set be identified with $\{\bot\}$.

Although tools reasoning logically about a script can cope without a definite enumeration order for sets, direct implementations of the operational semantics, such as that of Chapter 8, cannot. The model thus needs to specify some ordering on the production of elements from a set, making $\mathbb{P}(F)$ insufficient.

However, it is not necessary to specify the enumeration order in full detail. For example, given $\{n{*}2, n{*}3 \mid n \leftarrow \{1..\}\}$ we do not wish to specify if this resembles $\{2, 3, 4, 6, \ldots\}$, $\{3, 2, 4, 6, \ldots\}$ or any other enumeration where the elements at each stage are adjacent. This suggests that we might take

$$S_0 \quad = \quad \langle \mathbb{F}(F) \setminus \{\{\}\} \rangle$$

as the basic set representation; the terms are produced in finite, non-empty packets in the order given by the sequence, but the order within a packet is undefined. There is then a function $\varsigma : S_0 \to \mathbb{P}(F)$ which flattens the enumeration to a set and is given by

$$\begin{aligned} \varsigma(\langle\rangle) &= \{\} \\ \varsigma(\langle f \rangle \frown fs) &= f \cup \varsigma(fs) \end{aligned}$$

For some purposes, this representation is also insufficient. For example, if we consider the open range $x = \{5..\}$ then this maps naturally into S_0 as $\langle\{5\}, \{6\}, \ldots\rangle$. Testing for membership in this is inefficient, since we know that it can be performed with a simple comparison. In fact, although it is 'obvious' that $1 \notin x$, an attempt to determine this by enumerating the elements will not terminate. This suggests that we can achieve better results for membership tests if we model some set operations directly in the domain. These operations include pairwise union, intersection, and difference, as well as open and closed ranges. We thus take

$$\begin{aligned} S_0 &= \langle \mathbb{F}(F) \setminus \{\{\}\} \rangle \\ S &= S_0 + S \underline{\cup} S + S \underline{\cap} S + S \underline{\setminus} S + N \underline{..} + N \underline{..} N \end{aligned}$$

where the constructions such as $S \underline{\cup} S$ simply denote tagged composites from the appropriate product domain. More branches will be added to S later, but for now we return to the construction of E.

We take $V = \{error\}$, where the single error value indicates that the original expression was badly typed. The *error* value will therefore be regarded as serious and be propagated from any subexpressions. Any other errors will correspond to divergence and be propagated in a similar but less forceful way.

Combining these components, we take E to be the domain given by

$$E = (N + B + L + V + S + E \to E + \langle E \rangle + E^* + Name \times E)^{\bullet}$$

using similar notation here to denote domain constructors to that used for set operations earlier. The sets N, B, V and L are converted to domains simply by lifting the 'flat' set above a bottom element. The bottom elements are then combined using the (coalesced) sum operation. The final component is used for handling type definitions; see Section 3.12 for details.

Again, the dotted components of E are required to have no directly nested dot-structure, and this is enforced by making the outermost domain constructor be the dot-list construction. This means that all of the 'simple' values in E are actually singleton dot-lists; we will ignore this in practice, making implicit use of the projection from singletons to the value they contain.

This domain definition would allow us to model datatypes which had functions as components. However, events lie in Σ which forbids these, along with other 'infinite' values. For simplicity, we disallow functions as components in all datatypes, not just in events.

There is a natural embedding $\iota : F \to E$ given by

$$\begin{aligned}
\iota(n : N) &= n \\
\iota(b : B) &= b \\
\iota(l : L) &= l \\
\iota(\{e_1, \ldots e_n\}) &= \langle\{e_1, \ldots e_n\}\rangle \\
\iota(\langle e_1, \ldots e_n\rangle) &= \langle\iota(e_1), \ldots \iota(e_n)\rangle \\
\iota((e_1, \ldots e_n)) &= (\iota(e_1), \ldots \iota(e_n)) \\
\iota(e_1 \bullet \ldots e_n) &= \iota(e_1) \bullet \ldots \iota(e_n)
\end{aligned}$$

This embedding identifies F with elements in E which are both maximal and finite so that the function $\pi : E \to F_\perp$ given by

$$\pi(e) = \begin{cases} \iota^{-1}(e) & \text{, if } e \in \operatorname{ran}\iota \\ \perp & \text{, otherwise} \end{cases}$$

is well-defined and continuous.

3.2 Process domains

We now have a domain E which is sufficient to model the non-process components of the language. We have Σ, contained within E, from which we can construct the failures-divergences model $\mathcal{N}$. We must now combine these to give a domain $D_{\mathcal{N}}$ which will allow us to define the denotational semantics for the language as a whole.

We parameterise the domain D since later chapters will use alternative models for the process components.

The minimal choice when modeling processes and values is to take

$$D_X = E + X$$

but this does not allow process terms to take parameters. Since many useful examples make use of subscripted process names, this is clearly inadequate.

A better solution is to take

$$D_X = E + X + E \to X$$

so that processes can take parameters, but only expression parameters. This permits most common uses of subscripted processes, but does not allow the use of user-defined process operations such as

$$loop(P) \quad = \quad P \,;\, loop(P)$$

Despite this restriction, this domain is expressive enough for many cases, and for simplicity is used for the purposes of this document.

Other possibilities

At the cost of complicating many of the proofs it should be possible to allow

$$D_X \quad = \quad E + X + D_X^* + D_X \to X$$

which allows process parameters only to process-valued functions.

The larger domain

$$D_X \quad = \quad E + X + D_X^* + D_X \to D_X$$

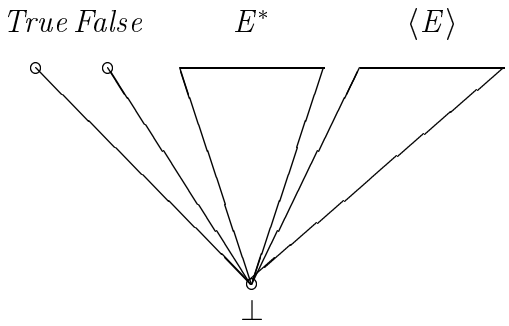
allows process parameters in general. In fact, this final alternative is too powerful for the language presented here since we currently have no operators which take processes arguments and yield a non-process result. (Note that this is one reason that the σ operation which maps from processes to their communication sets has not been provided.)

Those places where using one of the more complex domains would require changes to the semantics or proof will be noted.

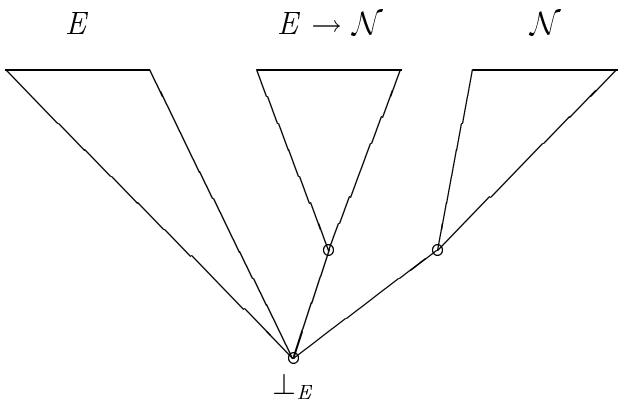
Combining the domains

The choice of the sum operation used to build the expression and process domains controls whether the $\perp$ values of the domains combine. The sum operation in domain E combines a number of flat set components such as B with a number of domains such as E^* and $\langle E \rangle$. The bottom elements of the domains are combined to form a common bottom element, which is shared with the bottom added to the sets when they are turned into flat domains. The $B + E^* + \langle E \rangle$ component thus

resembles



When combining the processes component of the model with the expression domain, we must decide whether to identify the bottom elements of the domains. From the operational semantics given later, we note that while a divergent process performs an infinite sequence of internal actions, a $\perp$ which occurs as part of the expression reduction involved in progressing a process will prevent us seeing any further actions; in some sense, the $\perp_E$ corresponds to $STOP$ rather than $\perp_{\mathcal{N}}$. This suggests that we should *not* combine these two bottom values and the domain sum in D_X should ensure this, so that $D_{\mathcal{N}}$ resembles



3.3 Environment

We take environments to be functions from names to values, so

$$Env_X \quad = \quad Name \to D_X$$

When generating environments there is the possibility of a type error so we take

$$Env_X^+ \quad = \quad Env_X \cup \{error\}$$

When combining values from Env^+ using the functional override operation we take

$$\begin{aligned} x \oplus error &= error \\ error \oplus x &= error \end{aligned}$$

and binding an identifier to the *error* value yields the error environment, so that

$$\{x \mapsto error\} = error$$

We define a semantic function

$$\mathcal{E} : Any \to Env^+ \to D$$

where Any is the space of all syntactic productions, and in all cases

$$\mathcal{E}[\![e]\!]error = error$$

When defining the semantic function, we will in general require the value of sub-terms to lie within specific parts of D. If the sub-terms do not lie within those parts indicated, we define the value returned by $\mathcal{E}$ to be *error*.

We define $\mathcal{E}$ over the grammar presented in Section 2.16. We take

$$\begin{aligned} n &: Number \\ e, e_i &: Expr \\ es &: Exprs \\ b &: Bool \\ \rho &: Env_{\mathcal{N}} \end{aligned}$$

throughout the following sections.

3.4 Booleans

The results of boolean expression lie in $B = \{True, False\}$, and we have

$$\begin{aligned} \mathcal{E}[\![\text{true}]\!]\rho &= True \\ \mathcal{E}[\![\text{false}]\!]\rho &= False \end{aligned}$$

To simplify the redundancy among the comparison operations, we define some in terms of others using

$$\begin{aligned} \mathcal{E}[\![e_1 \neq e_2]\!]\rho &= \mathcal{E}[\![\neg\ (e_1 = e_2)]\!]\rho \\ \mathcal{E}[\![e_1 > e_2]\!]\rho &= \mathcal{E}[\![e_2 < e_1]\!]\rho \\ \mathcal{E}[\![e_1 \geqslant e_2]\!]\rho &= \mathcal{E}[\![e_2 \leqslant e_1]\!]\rho \end{aligned}$$

We define equality and ordering on numbers in the obvious way by

$$\begin{array}{rcl}
\mathcal{E}[\![e_1 = e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho = \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 < e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho < \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 \leqslant e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \leqslant \mathcal{E}[\![e_2]\!]\rho
\end{array}$$

provided $\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho \in N$.

For sequences, equality can be defined within the expression language in terms of the fundamental sequence operations using

$$eq(x,y) \quad = \quad \left\{ \begin{array}{l} \text{if } null(x) \text{ then} \\ \ null(y) \\ \text{else} \\ \ \neg\ null(y) \wedge head(x) = head(y) \wedge eq(tail(x), tail(y)) \end{array} \right.$$

It is valid to define sequence equality in terms of a call to this a function using

$$\mathcal{E}[\![e_1 = e_2]\!]\rho \quad = \quad \mathcal{E}[\![eq(e_1, e_2)]\!]\rho$$

provided $\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho \in \langle E \rangle$.

Similarly, we use $\leqslant$ on sequences to denote sequence-prefix so that

$$\begin{array}{rcl}
\mathcal{E}[\![e_1 \leqslant e_2]\!]\rho & = & \mathcal{E}[\![pfix(e_1, e_2)]\!]\rho \\
\mathcal{E}[\![e_1 < e_2]\!]\rho & = & \mathcal{E}[\![pfix'(e_1, e_2)]\!]\rho
\end{array}$$

provided $\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho \in \langle E \rangle$ and where

$$pfix(x,y) \quad = \quad \left\{ \begin{array}{l} null(x) \\ \ \vee \\ (\neg\ null(y) \wedge head(x) = head(y) \wedge pfix(tail(x), tail(y))) \end{array} \right.$$

$$pfix'(x,y) \quad = \quad \left\{ \begin{array}{l} \neg\ null(y) \\ \ \wedge \\ (null(x) \vee (head(x) = head(y) \wedge pfix'(tail(x), tail(y)))) \end{array} \right.$$

Conditional expressions are lazy and evaluate the corresponding argument, unless the guard yields an error case (or bottom)

$$\mathcal{E}[\![\text{if } b \text{ then } e_1 \text{ else } e_2]\!]\rho \quad = \quad \left\{ \begin{array}{ll} \mathcal{E}[\![e_1]\!]\rho & \text{, if } \mathcal{E}[\![b]\!]\rho = True \\ \mathcal{E}[\![e_2]\!]\rho & \text{, if } \mathcal{E}[\![b]\!]\rho = False \\ \mathcal{E}[\![b]\!]\rho & \text{, otherwise} \end{array} \right.$$

provided $\mathcal{E}[\![b]\!]\rho \in B_\perp$.

The binary boolean operators are 'short-circuit'. All of the boolean operators are expressed in terms of the equivalent conditional expressions

$$\begin{aligned}
\mathcal{E}[\![b_1 \wedge b_2]\!]\rho &= \mathcal{E}[\![\text{if } b_1 \text{ then } b_2 \text{ else } b_1]\!]\rho \\
\mathcal{E}[\![b_1 \vee b_2]\!]\rho &= \mathcal{E}[\![\text{if } b_1 \text{ then } b_1 \text{ else } b_2]\!]\rho \\
\mathcal{E}[\![\neg\ b]\!]\rho &= \mathcal{E}[\![\text{if } b \text{ then false else true}]\!]\rho
\end{aligned}$$

provided $\mathcal{E}[\![b]\!]\rho, \mathcal{E}[\![b_1]\!]\rho, \mathcal{E}[\![b_2]\!]\rho \in B$.

3.5 Numbers

Taking *Number* as the space of numeric literals, we assume that N, the set modeling numbers, matches this space, and that there exists a function $numb : Number \to N$. We then take, for $n \in Number$,

$$\mathcal{E}[\![n]\!]\rho = numb(n)$$

Similarly, we assume that N provides arithmetic operations and take

$$\begin{aligned}
\mathcal{E}[\![e_1 + e_2]\!]\rho &= \mathcal{E}[\![e_1]\!]\rho + \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 - e_2]\!]\rho &= \mathcal{E}[\![e_1]\!]\rho - \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![-\ e]\!]\rho &= -\mathcal{E}[\![e]\!]\rho \\
\mathcal{E}[\![e_1 * e_2]\!]\rho &= \mathcal{E}[\![e_1]\!]\rho * \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 / e_2]\!]\rho &= \mathcal{E}[\![e_1]\!]\rho \,/\, \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 \,\%\, e_2]\!]\rho &= \mathcal{E}[\![e_1]\!]\rho \,\%\, \mathcal{E}[\![e_2]\!]\rho
\end{aligned}$$

provided $\mathcal{E}[\![e]\!]\rho, \mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho \in N$. If $\mathcal{E}[\![e_2]\!]\rho \neq 0$ in the last two definitions, then we need to generate a runtime error. The $error \in V$ value is unsuitable since that represents errors which are statically determinable. This can be achieved by adding values to V, but for simplicity we define the result in this case to be $\perp_E$.

We observe that implementations are likely to be much more useful if some guarantees are provided for these operations on N. Implementations are required to guarantee a minimum of 32-bit signed integers, with the effects of overflow undefined (including division by zero.) This permits both twos-complement and sign-bit representations.

In the case of the division operations we require that, for integers a and b,

$$\begin{aligned}
b \neq 0 &\Rightarrow 0 \leqslant |a\%b| < |b| \\
b \neq 0 &\Rightarrow (a/b) * b + a\%b = a \\
b > 0 &\Rightarrow 0 \leqslant a\%b
\end{aligned}$$

These are approximately the guarantees provided by ANSI-C for the '`long int`' type, although the behaviour of the remainder operator is more tightly defined than that guaranteed by C (see [17] for details.)

3.6 Tuples

Tuples map naturally into the E^* component provided for them, so that

$$\mathcal{E}[\![(e_1, \ldots e_n)]\!]\rho \quad = \quad (\mathcal{E}[\![e_1]\!]\rho, \ldots \mathcal{E}[\![e_n]\!]\rho)$$

provided $\mathcal{E}[\![e_i]\!]\rho \in E$.

3.7 Sequences

The sequence component of the domain is written $\langle E \rangle$ for the full-expression domain, and $\langle F \rangle_f$ for the finite version. The latter can of course be modeled in the same way as the tuples of F and the former is given by the domain equation

$$\langle E \rangle \quad = \quad E \times \langle E \rangle + \{nil\}$$

In this section, we will assume that elements of $\langle E \rangle$ can be decomposed in this way, so that for $x \in \langle E \rangle$ either $x = \perp$, or $x = nil$ the empty sequence, or $x = y : z$ for some $y \in E$ and $z \in \langle E \rangle$.

Simple sequence literals map directly into $\langle E \rangle$

$$\mathcal{E}[\![\langle e_1, \ldots e_n \rangle]\!]\rho \quad = \quad \mathcal{E}[\![e_1]\!]\rho : \ldots \mathcal{E}[\![e_n]\!]\rho : nil$$

provided $\mathcal{E}[\![e_i]\!]\rho \in E$.

The fundamental operations on sequences are those corresponding to the cons, car, cdr, and null operations of Lisp. A sequence is null if and only if it corresponds to nil so

$$\mathcal{E}[\![null(e)]\!]\rho \quad = \quad \begin{cases} \perp & \text{, if } \mathcal{E}[\![e]\!]\rho = \perp \\ True & \text{, if } \mathcal{E}[\![e]\!]\rho = nil \\ False & \text{, otherwise} \end{cases}$$

provided $\mathcal{E}[\![e]\!]\rho \in \langle E \rangle$.

The function *cons* adds a new element to a sequence in the usual way

$$\mathcal{E}[\![cons(e_1, e_2)]\!]\rho \quad = \quad \mathcal{E}[\![e_1]\!]\rho : \mathcal{E}[\![e_2]\!]\rho$$

and the *head* and *tail* operations reverse this operation

$$\begin{aligned}\mathcal{E}[\![head(e)]\!]\rho &= \begin{cases} x & \text{, if } \mathcal{E}[\![e]\!]\rho = x : y \\ \bot & \text{, otherwise} \end{cases} \\ \mathcal{E}[\![tail(e)]\!]\rho &= \begin{cases} y & \text{, if } \mathcal{E}[\![e]\!]\rho = x : y \\ \bot & \text{, otherwise} \end{cases}\end{aligned}$$

provided $\mathcal{E}[\![e]\!]\rho \in \langle E \rangle$.

Given these basic operations, catenation can be expressed as

$$\mathcal{E}[\![e_1 \frown e_2]\!]\rho = \mathcal{E}[\![cat(e_1, e_2)]\!]\rho$$

provided $\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho \in \langle E \rangle$ where

$$cat(x, y) = \text{if } null(x) \text{ then } y \text{ else } cons(head(x), cat(tail(x), y))$$

Similarly, open and closed ranges can be expressed simply using

$$\begin{aligned}\mathcal{E}[\![\langle e_1 .. e_2 \rangle]\!]\rho &= \mathcal{E}[\![closed(e_1, e_2)]\!]\rho \\ \mathcal{E}[\![\langle e_1 .. \rangle]\!]\rho &= \mathcal{E}[\![open(e_1)]\!]\rho\end{aligned}$$

where

$$\begin{aligned}closed(m, n) &= \text{if } m > n \text{ then } \langle\rangle \text{ else } cons(m, closed(m{+}1, n)) \\ open(n) &= cons(n, open(n{+}1))\end{aligned}$$

We define the semantic function on sequence comprehensions by induction on the number of generators. The base case (of no generators) is simply an unusual way of writing the literal.

$$\mathcal{E}[\![\langle es \mid \epsilon \rangle]\!]\rho = \mathcal{E}[\![\langle es \rangle]\!]\rho$$

For the other cases we use the functions *mapseq* and *concat* to eliminate the outermost generator from the comprehension, so that

$$\begin{aligned}\mathcal{E}[\![\langle es \mid e_1 \leftarrow e_2, gs \rangle]\!]\rho &= \mathcal{E}[\![concat(mapseq(\lambda\, e_1.\langle es \mid gs \rangle, e_2))]\!]\rho \\ \mathcal{E}[\![\langle es \mid b, gs \rangle]\!]\rho &= \mathcal{E}[\![\text{if } b \text{ then } \langle es \mid gs \rangle \text{ else } \langle\rangle]\!]\rho\end{aligned}$$

The auxiliary functions are given by

$$\begin{aligned}mapseq(f, s) &= \begin{cases} \text{if } null(s) \text{ then} \\ \quad \langle\rangle \\ \text{else} \\ \quad cons(f(head(s)), mapseq(f, tail(s))) \end{cases} \\ concat(s) &= \text{if } null(s) \text{ then } \langle\rangle \text{ else } head(s) \frown concat(tail(s))\end{aligned}$$

For the size of a list, we take

$$length(s) \;=\; \text{if } null(s) \text{ then } 0 \text{ else } 1 + length(tail(s))$$

and write

$$\mathcal{E}[\![\#e]\!]\rho \;=\; \mathcal{E}[\![length(e)]\!]\rho$$

provided $\mathcal{E}[\![e]\!]\rho \in \langle E \rangle$.
Similarly,

$$\mathcal{E}[\![e_1 \text{ in } e_2]\!]\rho \;=\; \mathcal{E}[\![elem(e_1, e_2)]\!]\rho$$

provided $\mathcal{E}[\![e]\!]\rho \in E$ and $\mathcal{E}[\![e_2]\!]\rho \in \langle E \rangle$ where

$$elem(x, s) \;=\; \neg\, null(s) \wedge (x = head(s) \vee elem(x, tail(s)))$$

3.8 Sets

Given the structure of S, the set component in the expression domain, many of the set operations have direct implementations

$$\begin{aligned}
\mathcal{E}[\![e_1 \cup e_2]\!]\rho &= x_1 \underline{\cup} x_2 \\
\mathcal{E}[\![e_1 \cap e_2]\!]\rho &= x_1 \underline{\cap} x_2 \\
\mathcal{E}[\![e_1 \setminus e_2]\!]\rho &= x_1 \underline{\setminus} x_2
\end{aligned}$$

provided $x_i = \mathcal{E}[\![e_i]\!]\rho \in S$,

$$\begin{aligned}
\mathcal{E}[\![\{\}]\!]\rho &= \langle\rangle \\
\mathcal{E}[\![\{e_1, \ldots e_n\}]\!]\rho &= \langle\{\pi(x_1), \ldots \pi(x_n)\}\rangle
\end{aligned}$$

where $x_i = \mathcal{E}[\![e_i]\!]\rho$, and

$$\begin{aligned}
\mathcal{E}[\![\{e_1..\}]\!]\rho &= x_1 \underline{..} \\
\mathcal{E}[\![\{e_1..e_2\}]\!]\rho &= x_1 \underline{..} x_2
\end{aligned}$$

provided $x_i = \mathcal{E}[\![e_i]\!]\rho \in N$.

We now define the functions on S which are required to define the remaining set operations. To simplify the discussion, the *error* value will not be mentioned; it is assumed to be propagated in the usual way.

Membership of a set may be tested using $member : (F, S) \to V + B$ where

$$member(x, a \underline{\cup} b) = \begin{cases} \bot & \text{, if } member(x,a) = \bot \\ True & \text{, if } member(x,a) = True \\ member(x,b) & \text{, if } member(x,a) = False \end{cases}$$

$$member(x, a \underline{\cap} b) = \begin{cases} \bot & \text{, if } member(x,a) = \bot \\ False & \text{, if } member(x,a) = False \\ member(x,b) & \text{, if } member(x,a) = True \end{cases}$$

$$member(x, a \underline{\setminus} b) = \begin{cases} \bot & \text{, if } member(x,a) = \bot \\ False & \text{, if } member(x,a) = False \\ \neg\ member(x,b) & \text{, if } member(x,a) = True \end{cases}$$

for the set operations. For ranges, we have

$$\begin{aligned} member(x, m \underline{..}) &= m \leqslant x \\ member(x, m \underline{..} n) &= m \leqslant x \wedge x \leqslant n \end{aligned}$$

provided $x \in N$. For the base case of membership on S_0, we simply examine the finite productions in order, stopping when we find the element, exhaust the supply, or produce a type error or bottom.

$$member(x, \langle\rangle) = False$$

$$member(x, \langle f \rangle \frown fs) = \begin{cases} \bot & \text{, if searching } f \text{ yields } \bot \\ True & \text{, if searching } f \text{ yields } True \\ member(x, fs) & \text{, if searching } f \text{ yields } False \end{cases}$$

Since every set has a representation (not unique) in S_0, there is obviously an enumeration function $\xi : S \to S_0$. We permit repeated elements in S_0, so enumeration of a union of two sets simply catenates the enumerations of the two components

$$\xi(x \underline{\cup} y) = \xi(x) \frown \xi(y)$$

If the first set has an infinite representation then the second set will effectively be hidden. This matches the left-to-right order used in the membership test above.

For intersections, we check each finite set in the enumeration of the first argument for membership in the second, so that

$$\xi(x \underline{\cap} y) = \mathit{filter}(\xi(x), y)$$

where

$$\mathit{filter}(\langle\rangle, y) = \langle\rangle$$

$$\mathit{filter}(\langle \{x_1, \ldots x_n\} \rangle \frown fs, y) = \begin{cases} \mathit{filter}(fs, y) & \text{, if } f' = \{\} \\ \langle f' \rangle \frown \mathit{filter}(fs, y) & \text{, otherwise} \end{cases}$$

where $f' = \bigcup_{1 \leqslant i \leqslant n} test(x_i, y)$ and

$$test(x, y) \;=\; \begin{cases} \{\bot\} & \text{, if } member(x, y) = \bot \\ \{x\} & \text{, if } member(x, y) = True \\ \{\} & \text{, if } member(x, y) = False \end{cases}$$

Set difference differs only in the negation of the final two guards in *test*.

Ranges simply count out the elements during the enumeration, so that

$$\begin{aligned} \xi(m\underline{..}) &= \langle\{m\}\rangle^\frown \xi((m{+}1)\underline{..}) \\ \xi(m\underline{..}n) &= \begin{cases} \langle\rangle & \text{, if } m > n \\ \langle\{m\}\rangle^\frown \xi((m{+}1)\underline{..}n) & \text{, otherwise} \end{cases} \end{aligned}$$

Given these operations, we can define distributed union and intersection

$$\mathcal{E}[\![\bigcup e]\!]\rho \;=\; union(\xi(x))$$

provided $x = \mathcal{E}[\![e]\!]\rho \in S$ and

$$\mathcal{E}[\![\bigcap e]\!]\rho \;=\; inter(\xi(x))$$

provided $x = \mathcal{E}[\![e]\!]\rho \in S$ and is non-empty ($\xi(x) \neq \langle\rangle$). The auxiliary functions simply build the appropriate element of S from the enumeration, so that

$$\begin{aligned} union(\langle\rangle) &= \langle\rangle \\ union(\langle\{x_1, \ldots x_n\}\rangle^\frown fs) &= \langle x_i \mid x_i \neq \{\}\rangle \underline{\cup} union(fs) \end{aligned}$$

provided $x_i \in \mathbb{F}(F)$, with *inter* defined similarly as

$$\begin{aligned} inter(\langle\{x_1, \ldots x_n\}\rangle) &= x_1 \underline{\cap} \ldots x_n \\ inter(\langle\{x_1, \ldots x_n\}\rangle^\frown fs) &= (x_1 \underline{\cap} \ldots x_n) \underline{\cap} inter(fs) \end{aligned}$$

Set comprehensions are unrolled into conditionals and distributed union operations in exactly the same way that sequence comprehensions use conditionals and concatenation, so that

$$\begin{aligned} \mathcal{E}[\![\{es \mid e_1 \leftarrow e_2, gs\}]\!]\rho &= \mathcal{E}[\![\bigcup mapset(\lambda\, e_1.\{es \mid gs\}, e_2)]\!]\rho \\ \mathcal{E}[\![\{es \mid b, gs\}]\!]\rho &= \mathcal{E}[\![\text{if } b \text{ then } \{es \mid gs\} \text{ else } \{\}]\!]\rho \end{aligned}$$

provided $\mathcal{E}[\![e_2]\!]\rho \in \{E\}$ and $\mathcal{E}[\![b]\!]\rho \in B$.

The required auxiliary function is

$$\mathcal{E}[\![mapset(e_1, e_2)]\!]\rho \;=\; map(x_1, \xi(x_2))$$

provided $x_1 = \mathcal{E}[\![e_1]\!]\rho \in E \to E$ and $x_2 = \mathcal{E}[\![e_2]\!]\rho \in S$, and where

$$\begin{array}{rcl} map(g, \langle\rangle) & = & \langle\rangle \\ map(g, \langle\{x_1, \ldots x_n\}\rangle^\frown fs) & = & \langle\{\pi(g(\iota(x_i))) \mid i \in \{1..n\}\}\rangle^\frown map(g, fs) \end{array}$$

Membership tests can be defined using the function defined earlier

$$\mathcal{E}[\![e_1 \in e_2]\!]\rho \;=\; member(\pi(x_1), x_2)$$

provided $x_1 = \mathcal{E}[\![e_1]\!]\rho \in E$ and $x_2 = \mathcal{E}[\![e_2]\!]\rho \in S$.

Cardinality can be defined directly using

$$\mathcal{E}[\![\mathrm{card}(e)]\!]\rho \;=\; card(x)$$

provided $x = \mathcal{E}[\![e]\!]\rho \in S$, where

$$\begin{array}{rcl} card(x \underline{\cup} y) & = & card(\xi(x \underline{\cup} y)) \\ card(x \underline{\setminus} y) & = & card(\xi(x \underline{\setminus} y)) \\ card(x \underline{\cap} y) & = & card(\xi(x \underline{\cap} y)) \end{array}$$

The remaining cases can be defined simply as

$$card(m \underline{..} n) \;=\; \begin{cases} 0 & \text{, if } m > n \\ 1 + (m - n) & \text{, if } m \leqslant n \end{cases}$$

and

$$\begin{array}{rcl} card(\langle\rangle) & = & 0 \\ card(\langle\{x_1, \ldots x_n\}\rangle^\frown fs) & = & m + card(fs \underline{\setminus} \langle\{x_1, \ldots x_n\}\rangle) \end{array}$$

where m is the number of distinct elements in $\bigcup_{1 \leqslant i \leqslant n} x_i$, which must be finite. Of course, $card(m \underline{..})$ cannot be finite.

Given the enumeration ξ from S to S_0, we can extend the definition of $\varsigma : S_0 \to \mathbb{P}(F)$ to $\varsigma : S \to \mathbb{P}(F)$ by

$$\varsigma(x) \;=\; \varsigma(\xi(x))$$

for $x \in S$, $x \notin S_0$.

3.9 Composites

The dot operation is used to form events and the elements of datatypes. As discussed during the construction of E and F, we regard the dot-lists as *flat*, with no nested dot structure; while such a structure is implied by arities which we associate with the labels in L, it is not represented directly. This is the simplest way to identify $(c.0).1$ with $c.(0.1)$.

Thus the dot operation is associative and of type $E \times E \to E^\bullet$ or $F \times F \to F^\bullet$ as appropriate. Given this,

$$\mathcal{E}[\![e_1.e_2]\!]\rho \;=\; \mathcal{E}[\![e_1]\!]\rho \bullet \mathcal{E}[\![e_2]\!]\rho$$

Essentially, the dot operation is simple catenation on dot-lists, with an automatic conversion from other values to singleton dot-lists if required.

Given a function *extensions* which yields the set of possible extensions for a given channel, the function *productions* which returns the full set of events associated with a channel can be defined as

$$\mathcal{E}[\![\mathit{productions}(e)]\!]\rho \;=\; \{\mathcal{E}[\![e]\!]\rho \bullet x \mid x \in \mathit{extensions}(\mathcal{E}[\![e]\!]\rho)\}$$

The details of *extensions* will be given later.

3.10 Binding

For lambda terms, the function arguments are bound on application so that

$$\mathcal{E}[\![\,\lambda\, es.e']\!]\rho \;=\; \lambda\, x.\mathcal{E}[\![e']\!](\rho \oplus \mathit{bind}((es), x))$$

where

$$\begin{aligned}
\mathit{bind}(e, \mathit{error}) &= \mathit{error} \\
\mathit{bind}(n : \mathit{Name}, x) &= \{n \mapsto x\} \\
\mathit{bind}((e_1, \ldots e_n), (x_1, \ldots x_n)) &= \bigcup_{1 \leqslant i \leqslant n} \rho_i \\
&\quad \text{where } \rho_i = \mathit{bind}(e_i, x_i)
\end{aligned}$$

provided the $\mathrm{dom}\,\rho_i$ are pairwise disjoint.

A given lambda term must lie in one of the two function components of the domain $D_\mathcal{N}$, these being $E \to E$ and $E \to \mathcal{N}$. There is a potential ambiguity here, but for a well-typed program the value must lie in precisely one of the two components; if it does not, then we can simply map the lambda term to the *error* value.

These lambda terms can then be used in applications, so that

$$\mathcal{E}[\![e(es)]\!]\rho \;=\; \mathit{apply}(\mathcal{E}[\![e]\!]\rho, \mathcal{E}[\![(es)]\!]\rho)$$

provided $\mathcal{E}[\![e]\!]\rho \in E \to E$ or $\mathcal{E}[\![e]\!]\rho \in E \to \mathcal{N}$. The function *apply* is the application function associated with values in $E \to E$.

3.11 Definitions

Names are simply looked up in the environment, so that for $n \in \mathit{Name}$

$$\mathcal{E}[\![n]\!]\rho \;=\; \rho(n)$$

provided $n \in \operatorname{dom}\rho$. The environment is extended using definitions (of values or types), either at the top level or within a 'let' clause.

A single definition binds a value to a one or more names. All definitions at the same level, either at the top level in a file or within a single 'let' block, are mutually recursive in the style of the Lisp `letrec` construct. To implement this we introduce semantic function $\mathcal{B}_{\mathcal{N}}$ which, given a definition and an environment, returns the effect it will have on that environment after a single level of unrolling. In the simplified case of a single, non-recursive definition this means that

$$\mathcal{E}[\![\text{let } d \text{ within } e]\!]\rho \;=\; \mathcal{E}[\![e]\!](\rho \oplus \mathcal{B}_{\mathcal{N}}[\![d]\!]\rho)$$

but for recursive definitions we need the standard approach of fixed points, so if d is a recursive definition we need to elaborate the above to

$$\begin{aligned}\mathcal{E}[\![\text{let } d \text{ within } e]\!]\rho &= \mathcal{E}[\![e]\!]\rho' \\ \rho' &= \rho \oplus \mathcal{B}_{\mathcal{N}}[\![d]\!]\rho'\end{aligned}$$

We consider it an error if a name is multiply defined; allowing alternative branches of a function to be selected by richer pattern-matching is considered in Section 11.1 as a possible extension.

Thus, the most general case of mutually recursive definitions is given by

$$\mathcal{E}[\![\text{let } d_1, \ldots d_n \text{ within } e]\!]\rho \;=\; \mathcal{E}[\![e]\!]\rho'$$

where

$$\rho' \;=\; \rho \oplus \bigcup_{1 \leqslant i \leqslant n} \mathcal{B}_{\mathcal{N}}[\![d_i]\!]\rho'$$

provided $\bigcup_{1 \leqslant i \leqslant n} \mathcal{B}_{\mathcal{N}}[\![d_i]\!]\rho'$ is functional.

The definition of the binding function is comparatively simple; for function definitions, we rewrite to the equivalent definition whose body is a lambda term

$$\mathcal{B}_{\mathcal{N}}[\![n(es) = e]\!]\rho \quad = \quad \mathcal{B}_{\mathcal{N}}[\![n = \lambda\, es.e]\!]\rho$$

Note that this permits curried definitions of the form $add(x)(y) = x + y$ with the above rule applied twice.

For all other cases we can make use of the mechanism for binding arguments used in the previous section

$$\mathcal{B}_{\mathcal{N}}[\![e_1 = e_2]\!]\rho \quad = \quad bind(e_1, \mathcal{E}[\![e_2]\!]\rho)$$

3.12 Type definitions

A type or channel definition identifies the tag names with values in L. The labels in L can in turn be looked up in the environment to yield the original tag and the extensions of the tag (the set of values which will from a complete value when dotted with the tag). We cannot store this information directly in L since this breaks the construction of F.

Type definitions also bind the name of the type to the set containing all elements of the type. Since these correspond to large, potentially infinite sets, it is useful to add more explicit set constructors to S to allow membership tests on these sets to circumvent some of the restrictions this might produce. The cardinality, membership, and enumeration operations on these constructors can be defined trivially.

We have

$$\begin{array}{rcl}
\mathit{TypeDef} & = & \text{datatype } \mathit{Name} = \mathit{VarType} \\
& | & \text{channel } \mathit{Names} : \mathit{Type} \\
& | & \text{channel } \mathit{Names} \\
\mathit{VarType} & = & \mathit{Branch} \,|\, \ldots \mathit{Branch} \\
\mathit{Branch} & = & \mathit{Name} \\
& | & \mathit{Name}.\mathit{Type} \\
\mathit{Type} & = & \mathit{Type} \\
& | & \mathit{Type}.\mathit{Type} \\
& | & (\mathit{Types}) \\
& | & \mathit{Expr} \\
& | & \mathit{Name} \\
& | & \mathit{Name}(\mathit{Types})
\end{array}$$

so the required set constructors correspond precisely to the choice connecting the branches, to the tagging in the branches, and to the dot product over types. We write these as $\underline{|}$, $\underline{.1}$ and $\underline{.}$ respectively. Given these, we define

$$\mathcal{B}_{\mathcal{N}}[\![\text{channel } c : t]\!]\rho \;=\; \{c \mapsto l, l \mapsto (c, \mathcal{E}[\![t]\!]\rho)\}$$

for some $l \in L$ which is not in the domain of ρ. For simple channels where no explicit protocol is given, we associate the label with the singleton set containing the identity value for the dot operation.

$$\mathcal{B}_{\mathcal{N}}[\![\text{channel } c]\!]\rho \;=\; \{c \mapsto l, l \mapsto (c, \{id\})\}$$

where id denotes this empty dotted value.

The sequential type t is expanded in the obvious way using

$$\mathcal{E}[\![t_1.t_2]\!]\rho \;=\; \mathcal{E}[\![t_1]\!]\rho \,\underline{.}\, \mathcal{E}[\![t_2]\!]\rho$$

Datatypes bind the tags in a similar way, but they also bind the typename, so that

$$\mathcal{B}_{\mathcal{N}}[\![\text{datatype } n = tag_1.t_1 \mid \ldots tag_n.t_n]\!]\rho \;=\; \begin{cases} \{n \mapsto tag_1 \,\underline{.1}\, x_1 \underline{|} \ldots tag_n \,\underline{.1}\, x_n\} \\ \cup \\ \bigcup_{1 \leqslant i \leqslant n} \{tag_i \mapsto l_i\} \\ \cup \\ \bigcup_{1 \leqslant i \leqslant n} \{l_i \mapsto (tag_i, \mathcal{E}[\![t_i]\!]\rho)\} \end{cases}$$

for some $l_i \in L$ which were not present in the domain of ρ and provided the typename and tags are all distinct. Simple branches with no explicit type component are handled as for simple channels.

The manipulation of events performed by the process operators, specifically the prefix operation '$\rightarrow$', requires a notion of the arity of a tag, which is the number of fields the tag expects to follow it. Since types are identified with the set of their elements, we define this operation on sets. It assumes that the sets involved are expressible as a rectangular product of simple sets. As an example, $\{1.3, 1.4, 2.3, 2.4\}$ is well-formed, but $\{1.3, 2.4\}$ is not.

The type arity function is given by

$$\begin{aligned} arity(tag \,\underline{.1}\, t) &= 1 \\ arity(t_1 \,\underline{.}\, t_2) &= arity(t_1) + arity(t_2) \\ arity(t_1 \underline{|} t_2) &= arity(t_1) \sqcup arity(t_2) \end{aligned}$$

and on other sets it suffices to choose an element and to count the fields within that using a similar function defined on values.

The built-in types Int and $Bool$, as well as the type constructors of Set, Seq, and tuples, also correspond to specialised branches of S, but the details of membership, cardinality, and enumeration are trivial and are omitted.

3.13 Processes

Before giving the semantic clauses for the process operators, we observe that the clauses so far are sufficient to yield values in the E component of D and in doing so they do not require any process-valued bindings in the environment, so that

$$\mathcal{E}[\![e]\!]\rho \in E \;\Rightarrow\; \mathcal{E}[\![e]\!]\rho = \mathcal{E}[\![e]\!](\rho \rhd E)$$

This makes the non-process semantics independent of the way in which we choose to define the process semantics; we will reuse $\mathcal{E}$ when defining the operational semantics of Chapter 4 and the tree semantics of Chapter 5. All of the process semantics depend upon $\mathcal{E}$ only as far as assuming the existence of E, $error \in E$, and of a suitable fixed point theory.

There are standard restrictions concerning unbounded non-determinism; to avoid introducing this, which requires more complex models than the simple failures-divergences model used here (see [31]), we are forced to restrict non-deterministic choice to be over finite sets of processes, and to limit hiding to finite sets of events. For simplicity later on, we also restrict the other replicated operators to be over finite sets or sequences.

As in [31], $\mathcal{N}$ is taken to be the set of all pairs (F, D) such that $F \subseteq \Sigma^* \times \mathbb{P}(\Sigma)$ and $D \subseteq \Sigma^*$ where for $s, t \in \Sigma^*$ and $X, Y \subseteq \Sigma$ we have

$$F \neq \{\} \tag{3.1}$$

$$(s^\frown t, \{\}) \in F \;\Rightarrow\; (s, \{\}) \in F \tag{3.2}$$

$$(t, X) \in F \wedge Y \subseteq X \;\Rightarrow\; (t, Y) \in F \tag{3.3}$$

$$(t, X) \in F \wedge (\forall\, a \in Y \bullet (t^\frown\langle a\rangle, \{\}) \notin F) \;\Rightarrow\; (t, X \cup Y) \in F \tag{3.4}$$

$$(\forall\, X' \in \mathbb{F}(X) \bullet (t, X') \in F) \;\Rightarrow\; (t, X) \in F \tag{3.5}$$

$$s \in D \;\Rightarrow\; s^\frown t \in D \tag{3.6}$$

$$s \in D \;\Rightarrow\; (s^\frown t, X) \in F \tag{3.7}$$

In addition, the semantic clauses are such that $\checkmark$ (which is a distinguished element of L) will be guaranteed to be final. This is enforced since *SKIP* is the only way $\checkmark$ can be generated and the parallel operations insist that $\checkmark$ is synchronised.

The semantic function $\mathcal{E}$ can now be defined on the elements $p : Proc$ by

$$\mathcal{E}[\![p]\!]\rho \;=\; (\mathcal{F}[\![p]\!]\rho, \mathcal{D}[\![p]\!]\rho)$$

so that $\mathcal{E}[\![p]\!]\rho \in \mathcal{N}$.

As usual, we define the traces and (immediate) refusals of a process to be given by

$$\mathcal{T}[\![p]\!]\rho \;=\; \{t \mid (t, \{\}) \in \mathcal{F}[\![p]\!]\rho\}$$
$$\mathcal{R}[\![p]\!]\rho \;=\; \{X \mid (\langle\rangle, X) \in \mathcal{F}[\![p]\!]\rho\}$$

Throughout the following definitions, we take

$$\begin{aligned} F &= \mathcal{F}[\![p]\!]\rho \\ D &= \mathcal{D}[\![p]\!]\rho \\ T &= \mathcal{T}[\![p]\!]\rho \\ F' &= \mathcal{F}[\![p']\!]\rho \\ D' &= \mathcal{D}[\![p']\!]\rho \end{aligned}$$

and require that $\mathcal{E}[\![p]\!]\rho, \mathcal{E}[\![p']\!]\rho \in \mathcal{N}$.

For the atomic processes we have

$$\begin{aligned} \mathcal{F}[\![STOP]\!]\rho &= \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \\ \mathcal{D}[\![STOP]\!]\rho &= \{\} \\ \mathcal{F}[\![SKIP]\!]\rho &= \{(\langle\rangle, X) \mid \surd \notin X \subseteq \Sigma\} \cup \{(\surd, X) \mid X \subseteq \Sigma\} \\ \mathcal{D}[\![SKIP]\!]\rho &= \{\} \end{aligned}$$

Choice

For the simple choice operators we have

$$\begin{aligned} \mathcal{F}[\![p \sqcap p']\!]\rho &= F \cup F' \\ \mathcal{D}[\![p \sqcap p']\!]\rho &= D \cup D' \\ \mathcal{F}[\![p \,\Box\, p']\!]\rho &= \left\{ \begin{array}{l} \{(\langle\rangle, X) \mid (\langle\rangle, X) \in F \cap F'\} \\ \cup \\ \{(s, X) \mid s \neq \langle\rangle, (s, X) \in F \cup F'\} \\ \cup \\ \{(s, X) \mid s \in \mathcal{D}[\![p \,\Box\, p']\!]\rho\} \end{array} \right. \\ \mathcal{D}[\![p \,\Box\, p']\!]\rho &= D \cup D' \end{aligned}$$

Sequential composition transfers control on the occurrence of $\surd$, which, by construction, is the only occurrence.

$$\begin{aligned} \mathcal{F}[\![p \,;\, p']\!]\rho &= \left\{ \begin{array}{l} \{(s, X) \mid (s, X \cup \{\surd\}) \in F, \neg \surd \text{ in } s\} \\ \cup \\ \{(s^\frown t, X) \mid s^\frown\langle\surd\rangle \in T, (t, X) \in F'\} \\ \cup \\ \{(s, X) \mid s \in \mathcal{D}[\![p \,;\, p']\!]\rho\} \end{array} \right. \\ \mathcal{D}[\![p \,;\, p']\!]\rho &= \left\{ \begin{array}{l} \{s^\frown t \mid s \in D, \neg \surd \text{ in } s\} \\ \cup \\ \{s^\frown t \mid s^\frown\langle\surd\rangle \in T, t \in D'\} \end{array} \right. \end{aligned}$$

The revised interrupt operator transfers control on an initial action of p', rather than a $\surd$ by p

$$\begin{aligned}
\mathcal{F}[\![p \triangle p']\!]\rho &= \left\{ \begin{array}{l} \{(s, X \cap Y) \,|(s, X) \in F, \neg \surd \text{ in } s, (\langle\rangle, Y) \in F'\} \\ \cup \\ \{(s, X) \,|(s, X) \in F, \surd \text{ in } s\} \\ \cup \\ \{(s^\frown t, X) \,|\, t \neq \langle\rangle, s \in T, \neg \surd \text{ in } s, (t, X) \in F'\} \\ \cup \\ \{(s, X) \,|\, s \in \mathcal{D}[\![p \triangle p']\!]\rho\} \end{array} \right. \\
\mathcal{D}[\![p \triangle p']\!]\rho &= D \cup \{s^\frown t \,|\, s \in T, \neg \surd \text{ in } s, t \in D'\}
\end{aligned}$$

Parallel

The alphabetised parallel operator has

$$\begin{aligned}
\mathcal{F}[\![p \;{}_e\|_{e'}\; p']\!]\rho &= \left\{ \begin{array}{l} \{(s, (X \cap A) \cup (Y \cap A') \cup Z) \,|\, Z \cap A'' = \{\}, \\ \qquad s \in A''^*, (s \upharpoonright A, X) \in F, (s \upharpoonright A', Y) \in F'\} \\ \cup \\ \{(s, X) \,|\, s \in \mathcal{D}[\![p \;{}_e\|_{e'}\; p']\!]\rho\} \end{array} \right. \\
\mathcal{D}[\![p \;{}_e\|_{e'}\; p']\!]\rho &= \left\{ \begin{array}{l} \{s^\frown t \,|\, s \in A''^*, s \upharpoonright A \in D, s \upharpoonright A' \in T'\} \\ \cup \\ \{s^\frown t \,|\, s \in A''^*, s \upharpoonright A' \in D', s \upharpoonright A \in T\} \end{array} \right.
\end{aligned}$$

where $x = \mathcal{E}[\![e]\!]\rho$, $x' = \mathcal{E}[\![e']\!]\rho$, $A = \varsigma(x)_\surd$, $A' = \varsigma(x')_\surd$, $A'' = A \cup A'$ and provided x and x' denote communication sets. If either of the sets evaluate to *error* or $\perp$, then that is the result of the process clause, with *error* worse than $\perp$ as usual. This approach will be also be used for the remaining process operators where appropriate.

The communicating parallel operator is similar

$$\begin{aligned}
\mathcal{F}[\![p \underset{e}{\|} p']\!]\rho &= \left\{ \begin{array}{l} \{(u, X \underset{A}{\|} Y) \,|\, u \in s \underset{A}{\|} t, (s, X) \in F, (t, Y) \in F'\} \\ \cup \\ \{(s, X) \,|\, s \in \mathcal{D}[\![p \underset{e}{\|} p']\!]\rho\} \end{array} \right. \\
\mathcal{D}[\![p \underset{e}{\|} p']\!]\rho &= \left\{ \begin{array}{l} \{u \,|\, u \in s \underset{A}{\|} t, s \in D, t \in T'\} \\ \cup \\ \{u \,|\, u \in s \underset{A}{\|} t, s \in T, t \in D'\} \end{array} \right.
\end{aligned}$$

where $x = \mathcal{E}[\![e]\!]\rho$, $A = \varsigma(x)_{\surd}$ and provided x is a communication set.

$$u \in s \underset{A}{\|} t \quad \Leftrightarrow \quad \left\{ \begin{array}{l} u = s = t = \langle\rangle \\ \vee \\ u_0 = s_0 \notin A \wedge u' \in s' \underset{A}{\|} t \\ \vee \\ u_0 = t_0 \notin A \wedge u' \in s \underset{A}{\|} t' \\ \vee \\ u_0 = s_0 = t_0 \in A \wedge u' \in s' \underset{A}{\|} t' \end{array} \right.$$

$$X \underset{A}{\|} Y \quad = \quad ((X \cup Y) \cap A) \cup ((X \cap Y) \setminus A)$$

For interleaving, we simply reuse the sharing operations

$$\mathcal{F}[\![p \,|||\, p']\!]\rho \quad = \quad \left\{ \begin{array}{l} \{(u, X \underset{\{\surd\}}{\|} Y) \mid u \in s \underset{\{\surd\}}{\|} t, (s, X) \in F, (t, Y) \in F'\} \\ \cup \\ \{(s, X) \mid s \in \mathcal{D}[\![p \,|||\, p']\!]\rho\} \end{array} \right.$$

$$\mathcal{D}[\![p \,|||\, p']\!]\rho \quad = \quad \left\{ \begin{array}{l} \{u \mid u \in s \underset{\{\surd\}}{\|} t, s \in D, t \in T'\} \\ \cup \\ \{u \mid u \in s \underset{\{\surd\}}{\|} t, s \in T, t \in D'\} \end{array} \right.$$

Renaming and hiding

For the renaming operators, we form the explicit component of the relation, representing it as a set of pairs of events.

$$\begin{array}{lcl} \mathcal{F}[\![p \,[\![ss \mid gs]\!]\,]\!]\rho & = & \{(s, X) \mid (t, Y) \in F, s \simeq_R t, X \simeq_R Y\} \\ \mathcal{D}[\![p \,[\![ss \mid gs]\!]\,]\!]\rho & = & \{s \mid t \in D, s \simeq_R t\} \end{array}$$

where $\simeq_R$ is the natural promotion of the relation given by

$$x \simeq_R y \quad \Leftrightarrow \quad (x, y) \in R \vee (x = y \wedge x \notin \mathrm{dom}\, R)$$

and where $R = \mathcal{E}[\![\{ss \mid gs\}]\!]\rho$ is required to denote a set of pairs of events which we model within S. To prevent the introduction of infinite non-determinism, we must restrict the relations to those where the inverse image of each point in Σ under the relation is a finite set.

The simplified form of the renaming operator is defined using the base case of the full form

$$\mathcal{E}[\![p \,[\![ss]\!]\,]\!]\rho \quad = \quad \mathcal{E}[\![p \,[\![ss \mid \epsilon]\!]\,]\!]\rho$$

For hiding, we simply remove the required events from traces, taking care since an unbounded sequence of such events will yield divergence.

$$\mathcal{F}[\![p \setminus e]\!]\rho = \begin{cases} \{(s \setminus A, X) \,|\, (s, X \cup A) \in \mathcal{F}[\![p]\!]\rho\} \\ \cup \\ \{(s, X) \,|\, s \in \mathcal{D}[\![p \setminus e]\!]\rho\} \end{cases}$$

$$\mathcal{D}[\![p \setminus e]\!]\rho = \begin{cases} \{(s \setminus A)^\frown t \,|\, s \in \mathcal{D}[\![p]\!]\rho\} \\ \cup \\ \{(s \setminus A)^\frown t \,|\, \forall\, n \in \mathbb{N} \bullet \exists\, u \in A^n \bullet s^\frown u \in T\} \end{cases}$$

provided $A = \varsigma(\mathcal{E}[\![e]\!]\rho)$ is a finite set of events.

Replicated operations

Since we are restricting replicated operators to finite control expressions, we can express them indirectly by unrolling them into the binary form. This is much simpler than writing direct expressions for some of the operations, especially in the operational semantics.

So, for example

$$\mathcal{E}[\![\Box\, e : e' \bullet p]\!]\rho = \mathcal{E}[\![p]\!]\rho_1 \,\Box\, \ldots \mathcal{E}[\![p]\!]\rho_n$$

$$\mathcal{E}[\![\sqcap\, e : e' \bullet p]\!]\rho = \mathcal{E}[\![p]\!]\rho_1 \sqcap \ldots \mathcal{E}[\![p]\!]\rho_n$$

where $\varsigma(\mathcal{E}[\![e']\!]\rho) = \{x_1, \ldots x_n\}$ and $\rho_i = \rho \oplus bind(e, x_i)$. For the non-deterministic choice operation we require that $n > 0$.

Similarly

$$\mathcal{E}[\![;\, e : e' \bullet p]\!]\rho = \mathcal{E}[\![p]\!]\rho_1 \,;\, \ldots \mathcal{E}[\![p]\!]\rho_n$$

where $\mathcal{E}[\![e']\!]\rho = \langle x_1, \ldots x_n \rangle$ and $\rho_i = \rho \oplus bind(e, x_i)$.

For the parallel operator we must build up the intermediate event sets

$$\mathcal{E}[\![\,\|\, e : e' \bullet (e'', p)]\!]\rho = p_1 \;{}_{a_1}\|_{\bigcup_{i \geqslant 2} a_i}\; (p_2 \;{}_{a_2}\| \ldots p_n)$$

where $\varsigma(\mathcal{E}[\![e']\!])\rho = \{x_1, \ldots x_n\}$, $\rho_i = \rho \oplus bind(e, x_i)$, $p_i = \mathcal{E}[\![p]\!]\rho_i$ and $a_i = \mathcal{E}[\![e'']\!]\rho_i$.

Prefixing

For prefixing, the syntax is given by

$$Proc = Chan\ Fields \rightarrow Proc$$

where

$$Field \;=\; ?Expr : Expr \;\mid\; ?Expr \;\mid\; !Expr$$

In general, input fields declare new variables, and these variables may be used to control fields further to the right, as in $c?x : \{0,1\}?y : \{x+1, x+2\} \rightarrow STOP$. To define the meaning of the prefix operator, we first construct the set of events offered by the process and then construct an environment corresponding to each event offered. Those environments are used to evaluate the body of the operator. Assuming we have two functions

$$\begin{array}{rcl} \mathit{offer} & : & (Chan, Fields, Env) \rightarrow \mathbb{P}(E^\bullet) + V \\ \mathit{after} & : & (Chan, Fields, E^\bullet, Env) \rightarrow Env \end{array}$$

we can define

$$\begin{array}{rcl} \mathcal{F}[\![c\ fs \rightarrow p]\!]\rho & = & \left\{ \begin{array}{l} \{(\langle\rangle, X) \mid X \cap A = \{\}\} \\ \cup \\ \{(\langle a\rangle^\frown s, X) \mid a \in A, (s, X) \in \mathcal{F}[\![p]\!]\rho_a\} \end{array} \right. \\ \mathcal{D}[\![c\ fs \rightarrow p]\!]\rho & = & \{\langle a\rangle^\frown s \mid a \in A, s \in \mathcal{D}[\![p]\!]\rho_a\} \end{array}$$

where

$$\begin{array}{rcl} A & = & \mathit{offer}[\![c\ fs]\!]\rho \\ \rho_a & = & \mathit{after}(c\ fs, a, \rho) \end{array}$$

As this shows, *offer* and *after* are precisely the functions required to move from the new, more detailed communication syntax to the more traditional CSP $a : A \rightarrow P$ form.

The function *offer* is calculated by walking over the fields and events offered by c

$$\begin{array}{rcl} \mathit{offer}[\![c\ fs]\!]\rho & = & \{\pi(\mathcal{E}[\![c]\!]\rho) \bullet \pi(e) \mid e \in \mathit{offer}'(fs, \varsigma(\mathit{extensions}(\mathcal{E}[\![c]\!]\rho)), \rho)\} \\ \mathit{offer}' & : & (Fields, \mathbb{P}(E^\bullet), Env) \rightarrow \mathbb{P}(E^\bullet) + V \end{array}$$

To simplify the definition of *offer'*, we first define, for $x \in E^\bullet$ and $S \in \mathbb{P}(E^\bullet)$

$$\begin{array}{rcl} S - x & = & \{y \in E^\bullet \mid x \bullet y \in S\} \\ x \leqslant_\rho S & \Leftrightarrow & S - x \neq \{\} \wedge x \text{ is well-formed with respect to } \rho \end{array}$$

where x is well-formed (with respect to ρ) if and only if it corresponds to a single component of a datatype, so that, given

$$datatype\ T \;=\; A.Bool | B.Int.Int$$

we regard A.false and B.0.0 as well-formed, but A, B.0, and A.false.A.true as not being well-formed. More formally, x is well-formed iff $wf(x, \rho)$ and $values(x, \rho) = 1$ where

$$wf(x, \rho) \quad \Leftrightarrow \quad (\forall\, y, z \bullet (x = y.z \wedge x \neq y) \Rightarrow values(y, \rho) \leqslant 0)$$

and

$$\begin{aligned} values(x_1 \bullet \ldots x_n, \rho) &= values(x_1, \rho) + \ldots values(x_n, \rho) \\ values(x, \rho) &= \begin{cases} 1 - arity(extensions(\rho(x))) & \text{, if } x \in L \\ 1 & \text{, if } x \notin L \end{cases} \end{aligned}$$

and $arity : S \to \mathbb{N}$ was defined earlier with type definitions.

As usual, we assume that *error* is propagated back up through the following definitions

$$\begin{aligned} offer'(\epsilon, S, \rho) &= \begin{cases} S & \text{, if } S \text{ contains only the empty dot-list} \\ error & \text{, otherwise} \end{cases} \\ offer'(!e\ fs, S, \rho) &= \{x \bullet y \mid y \in offer'(fs, S - x, \rho)\} \end{aligned}$$

where $x = \mathcal{E}[\![e]\!]\rho$ and provided $x \leqslant_\rho S$. For unrestricted inputs, we simply try binding each of the possible initial values in turn.

$$offer'(?e\ fs, S, \rho) = \begin{cases} \{x \bullet y \mid x \leqslant_\rho S, \rho' = bind(e, x), \\ \qquad y \in offer'(fs, S - x, \rho \oplus \rho')\} \end{cases}$$

Similarly, for restricted inputs we range over the set of possible values, with the side condition that they must all be permitted. We thus have

$$offer'(?e_1 : e_2\ fs, S, \rho) = \begin{cases} \{x \bullet y \mid x \in \mathcal{E}[\![e_2]\!]\rho, \rho' = bind(e_1, x), \\ \qquad y \in offer'(fs, S - x, \rho \oplus \rho')\} \end{cases}$$

provided $\forall\, x \in \mathcal{E}[\![e_2]\!]\rho \bullet x \leqslant_\rho S$.

As a special case, to allow the use of generic buffers, we permit trailing inputs to match multiple components, violating the well-formed condition. They will however consist of a dot-joined number of well-formed values, since all earlier fields were checked

$$\begin{aligned} offer'(?e_1 : e_2, S, \rho) &= \{x \mid x \in \mathcal{E}[\![e_2]\!]\rho\} \\ offer'(?e, S, \rho) &= S \end{aligned}$$

provided S contains only non-empty dot-chains and $\mathcal{E}[\![e_2]\!]\rho \subseteq S$.

We can now define *after* whose second argument is drawn from the offer set.

$$\begin{array}{rcl} after(c\ fs, a, \rho) & = & after'(fs, a - \mathcal{E}[\![c]\!]\rho, \rho) \\ after' & : & (Fields \times E^{\bullet} \times Env) \to Env \end{array}$$

and

$$\begin{array}{rcl} after'(\epsilon, y, \rho) & = & \rho \\ after'(!e\ fs, y, \rho) & = & after'(fs, y - \mathcal{E}[\![e]\!]\rho, \rho) \\ after'(?e\ fs, y, \rho) & = & after'(fs, y - x, \rho \oplus bind(e, x)) \\ after'(?e_1 : e_2\ fs, y, \rho) & = & after'(fs, y - x, \rho \oplus bind(e_1, x)) \end{array}$$

where x is taken to be the (unique) value in $E^{\bullet}$ satisfying $x \leqslant_{\rho} \{y\}$. The value $y - x$ is similarly taken to be the value in $\{y\} - x$. Both of these are well-defined provided y comes from the offer set associated with fs.

Chaos

We now define the meaning of *CHAOS*; it is unfortunate that there are two common meanings for this within the CSP community. The first, found in [18], is to use CHAOS to denote the worst-possible process, that is, the process $\perp_{\mathcal{N}}$. The other, as in [7] and [13], claims that this process already has a name in $\perp_{\mathcal{N}}$ and uses *CHAOS* to denote the most non-deterministic process which does not diverge. We follow the second, more recent, approach here and so have that

$$\begin{array}{rcl} \mathcal{F}[\![CHAOS(e)]\!]\rho & = & \left\{ \begin{array}{l} A^* \times \mathbb{P}(A) \\ \cup \\ \{s^\frown\langle\surd\rangle \mid s \in A^*\} \times \mathbb{P}(A_\surd) \end{array} \right. \\ \mathcal{D}[\![CHAOS(e)]\!]\rho & = & \{\} \end{array}$$

provided $A = \varsigma(\mathcal{E}[\![e]\!]\rho)$ is a finite set of events.

3.14 Summary

This chapter has taken the failures-divergences model and used it as the basis of a denotational semantics for the languageintroduce in Chapter 2. The semantics layers the process component on top of a common expression semantics which is be shared with the operational semantics in the next chapter.

Chapter 4

Operational semantics

We now build a hybrid operational semantics, where process semantics are given by a transition system in the style of Plotkin (see [28]), while the expression semantics use the denotational model already presented. The transition system is based upon that of [31] and [7], but has been adapted to the more detailed syntax used here.

The approach taken in [31], is to identify the closed terms of the syntax of CSP as configurations, and then to define transitions between these configurations. Such closed terms exist in the language used there, since it has no explicit notion of process parameterisation; recursion is handled explicitly using the fixed point notation, $\mu X \bullet P$.

We cannot use closed terms as the configurations for the language presented here. In general, a production of the syntax can have many free identifiers associated with it, values being bound to these identifiers in an environment. Those values will correspond either to expression values drawn from the domain E, or to process values which also correspond to configurations. We therefore use closures (pairs formed from process syntax and an environment) as the basis of our configurations

$$C = Proc \times Env_C$$

where

$$\begin{aligned} Env_C &= Name \to D_C \\ D_C &= E + C + E \to C \end{aligned}$$

However, C is not sufficient to define an operational semantics. The initial configuration of the interleaving of two processes can be represented as $(p \mid\mid\mid q, \phi)$ for some $p, q \in Proc$ and $\phi \in Env_C$, but if we consider the transition rules for that construct, we have

$$\frac{(p,\phi) \xrightarrow{\tau} (p',\phi')}{(p \;|||\; q,\phi) \xrightarrow{\tau} X}$$

where X is a configuration representing the interleaving of p' and q in ϕ' and ϕ respectively. Since ϕ and ϕ' may differ, this configuration need not lie within C.

The full space of configurations, Γ, thus includes C, as well as productions corresponding to most process operators and a distinguished *error* configuration

$$\begin{array}{rcl}
\Gamma & = & C \\
& | & \mathit{error} \\
& | & \Gamma \,\Box\, \Gamma \\
& | & \Gamma \,\triangle\, \Gamma \\
& | & \Gamma \,;\, C \\
& | & \Gamma \,|||\, \Gamma \\
& | & \Gamma \underset{A}{\|} \Gamma \\
& | & \Gamma \,{}_A\|_B\, \Gamma \\
& | & \Gamma \setminus A \\
& | & \Gamma \,[\![R]\!] \\
& | & \mathit{CHAOS}(A)
\end{array}$$

where $A, B \in \mathbb{P}(\Sigma)$ and $R \in \mathbb{P}(\Sigma \times \Sigma)$. The configuration X required above can then be written as $(p,\phi') \;|||\; (q,\phi) \in \Gamma$.

We augment the event space Σ to form $\Sigma_\tau = \Sigma \cup \{\tau\}$, whose elements are used to label the transitions, with the distinguished event τ used to mark internal transitions. The transition relation is then

$$\rightarrow \;\subseteq\; \Gamma \times \Sigma_\tau \times \Gamma$$

Throughout what follows, we take $p, q \in \mathit{Proc}$; $P, Q \in \Gamma$; $\phi \in \mathit{Env}_C$; $a \in \Sigma$; and $x \in \Sigma_\tau$. (It suffices to use Env_C rather than Env_Γ since identifiers are only bound to elements of D_C.)

The process *STOP* is trivial, having no transition rules at all, while *SKIP* is allowed to perform only $\surd$.

$$\frac{}{(\mathit{SKIP},\phi) \xrightarrow{\surd} (\mathit{STOP},\phi)}$$

4.1 Choice

Non-deterministic choice simply allows either alternative after an internal action

$$\frac{}{(p \sqcap q, \phi) \xrightarrow{\tau} (p, \phi)} \qquad \frac{}{(p \sqcap q, \phi) \xrightarrow{\tau} (q, \phi)}$$

External choice may require that progress is made inside the component processes for the decision can be resolved. To allow this we use the relevant component of Γ to push the environment down into the component branches.

If we assume that the operator has been promoted from a simple syntactic term to one using the operator in Γ, we have (as in [31])

$$\frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \,\Box\, Q \xrightarrow{\tau} P \,\Box\, Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P \,\Box\, Q \xrightarrow{a} Q'}$$

and the promotion is carried out using one level of 'lookahead' to avoid the introduction of 'extra' internal actions

$$\frac{(p, \phi) \xrightarrow{\tau} P'}{(p \,\Box\, q, \phi) \xrightarrow{\tau} P' \,\Box\, (q, \phi)} \qquad \frac{(q, \phi) \xrightarrow{\tau} Q'}{(p \,\Box\, q, \phi) \xrightarrow{\tau} (p, \phi) \,\Box\, Q'}$$

$$\frac{(p, \phi) \xrightarrow{a} P'}{(p \,\Box\, q, \phi) \xrightarrow{a} P'} \qquad \frac{(q, \phi) \xrightarrow{a} Q'}{(p \,\Box\, q, \phi) \xrightarrow{a} Q'}$$

4.2 Sequence

For sequential composition we allow the left component to carry out all actions up to the tick

$$\frac{P \xrightarrow{x} P'}{P \,;\, Q \xrightarrow{x} P' \,;\, Q} x \neq \checkmark \qquad \frac{P \xrightarrow{\checkmark} P'}{P \,;\, Q \xrightarrow{\tau} Q}$$

and promoting

$$\frac{(p, \phi) \xrightarrow{x} P'}{(p \,;\, q, \phi) \xrightarrow{x} P' \,;\, (q, \phi)} x \neq \checkmark \qquad \frac{(p, \phi) \xrightarrow{\checkmark} P'}{(p \,;\, q, \phi) \xrightarrow{\tau} (q, \phi)}$$

4.3 Interrupt

The new interrupt operation is similar to the external choice and sequential composition operator, but allows the right hand process to control the transfer. It is also necessary to prevent the transfer once the left hand process has terminated.

$$\frac{P \xrightarrow{x} P'}{P \triangle Q \xrightarrow{x} P' \triangle Q} x \neq \checkmark \qquad \frac{P \xrightarrow{\checkmark} P'}{P \triangle Q \xrightarrow{\checkmark} P'}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'} \qquad \frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'}$$

and promoting

$$\frac{(p,\phi) \xrightarrow{x} P'}{(p \triangle q,\phi) \xrightarrow{x} P' \triangle (q,\phi)} x \neq \checkmark \qquad \frac{(p,\phi) \xrightarrow{\checkmark} P'}{(p \triangle q,\phi) \xrightarrow{\checkmark} P'}$$

$$\frac{(q,\phi) \xrightarrow{\tau} Q'}{(p \triangle q,\phi) \xrightarrow{\tau} (p,\phi) \triangle Q'} \qquad \frac{(q,\phi) \xrightarrow{a} Q'}{(p \triangle q,\phi) \xrightarrow{a} Q'}$$

4.4 Interleave

Interleaving is simple

$$\frac{P \xrightarrow{x} P'}{P \,|||\, Q \xrightarrow{x} P' \,|||\, Q} x \neq \checkmark \qquad \frac{Q \xrightarrow{x} Q'}{P \,|||\, Q \xrightarrow{x} P \,|||\, Q'} x \neq \checkmark$$

$$\frac{P \xrightarrow{\checkmark} P' \wedge Q \xrightarrow{\checkmark} Q'}{P \,|||\, Q \xrightarrow{\checkmark} P' \,|||\, Q'}$$

with promotions

$$\frac{(p,\phi) \xrightarrow{x} P'}{(p \,|||\, q,\phi) \xrightarrow{x} P' \,|||\, (q,\phi)} x \neq \checkmark \qquad \frac{(q,\phi) \xrightarrow{x} Q'}{(p \,|||\, q,\phi) \xrightarrow{x} (p,\phi) \,|||\, Q'} x \neq \checkmark$$

$$\frac{(p,\phi) \xrightarrow{\checkmark} P' \wedge (q,\phi) \xrightarrow{\checkmark} Q'}{(p \,|||\, q,\phi) \xrightarrow{\checkmark} P' \,|||\, Q'}$$

4.5 Expressions

The non-process components are evaluated using a semantic function $\mathcal{C} : Any \to Env_C \to D_C$. On expressions, this can be defined using clauses identical to those used in the definition of $\mathcal{E}$, but on processes it yields process/environment pairs in C. Thus for $p \in Proc$

$$\mathcal{C}[\![p]\!]\phi \;=\; (p, \phi)$$

and similarly for $es \in Exprs$ and $p \in Proc$

$$\mathcal{C}[\![\,\lambda\, es.p]\!]\phi \;=\; \lambda\, x.(p, \phi \oplus bind((es), x))$$

Since the value of a non-process expression does not depend upon that of any process, we have that

$$\mathcal{C}[\![e]\!]\phi \in E \;\Rightarrow\; \mathcal{C}[\![e]\!]\phi = \mathcal{E}[\![e]\!]\check{\phi}$$

where $\check{\phi} = \phi \rhd E \in Env_{\mathcal{N}}$ excludes processes from the environment. This follows from the fact that the defining clauses for $\mathcal{C}$ and $\mathcal{E}$ on expressions are identical.

For some configurations in Γ, the evaluation of the expressions required as part of the operational semantics will yield $\perp$. We denote the set of all such configurations, $\Gamma_\perp$; if any subcomponent of a configuration lies within $\Gamma_\perp$ then the configuration lies in $\Gamma_\perp$.

The configurations which lie within $\Gamma_\perp$ correspond to the terms which diverge explicitly because of a problem in expression evaluation. The deduction rules given here should be taken as applying only to configurations outside $\Gamma_\perp$; we regard the configurations in $\Gamma_\perp$ as broken and final. As a result, the deduction rules will assume that expression evaluation does not fail.

4.6 Communicating parallel

When the semantic function $\mathcal{C}$ is applied to an expression as part of a process expansion from a non-broken configuration, there are three possible outcomes. If the result is indeed a set of events then all is well and the configuration can be replaced by the promoted form. If the result is the special *error* value, then we step immediately to the special *error* configuration in Γ. If the result is none of these then it is a value of the wrong type, so we again move to the *error* configuration. (As discussed above, if $\perp$ occurs then the configuration is broken.) Thus, if

$$A \;=\; \varsigma(\mathcal{C}[\![e]\!]\phi)$$

then we may handle all the error cases using

$$\frac{A \notin \mathbb{P}(\Sigma)}{(p \underset{e}{\parallel} q, \phi) \xrightarrow{\tau} error}$$

and the successful cases with

$$\frac{A \in \mathbb{P}(\Sigma) \wedge (p, \phi) \xrightarrow{x} P'}{(p \underset{e}{\parallel} q, \phi) \xrightarrow{x} P' \underset{A}{\parallel} (q, \phi)} x \notin A_\surd \qquad \frac{A \in \mathbb{P}(\Sigma) \wedge (q, \phi) \xrightarrow{x} Q'}{(p \underset{e}{\parallel} q, \phi) \xrightarrow{x} (p, \phi) \underset{A}{\parallel} Q'} x \notin A_\surd$$

$$\frac{A \in \mathbb{P}(\Sigma) \wedge (p, \phi) \xrightarrow{a} P' \wedge (q, \phi) \xrightarrow{a} Q'}{(p \underset{e}{\parallel} q, \phi) \xrightarrow{a} P' \underset{A}{\parallel} Q'} a \in A_\surd$$

and once the promotion has happened

$$\frac{P \xrightarrow{x} P'}{P \underset{A}{\parallel} Q \xrightarrow{x} P' \underset{A}{\parallel} Q} x \notin A_\surd \qquad \frac{Q \xrightarrow{x} Q'}{P \underset{A}{\parallel} Q \xrightarrow{x} P \underset{A}{\parallel} Q'} x \notin A_\surd$$

$$\frac{P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \underset{A}{\parallel} Q'} a \in A_\surd$$

4.7 Alphabetised parallel

Similarly for alphabetised parallel if we have

$$\begin{aligned} A &= \varsigma(\mathcal{C}[\![e_1]\!]\phi) \\ B &= \varsigma(\mathcal{C}[\![e_2]\!]\phi) \end{aligned}$$

then the error case is

$$\frac{A \notin \mathbb{P}(\Sigma) \vee B \notin \mathbb{P}(\Sigma)}{(p \;{}_{e_1}\|_{e_2}\; q, \phi) \xrightarrow{\tau} error}$$

with the successful cases

$$\frac{A, B \in \mathbb{P}(\Sigma) \wedge (p, \phi) \xrightarrow{x} P'}{(p \;{}_{e_1}\|_{e_2}\; q, \phi) \xrightarrow{x} P' \;{}_A\|_B\; (q, \phi)} x \in A_\tau \setminus B$$

$$\frac{A, B \in \mathbb{P}(\Sigma) \wedge (q, \phi) \xrightarrow{x} Q'}{(p \;{}_{e_1}\|_{e_2}\; q, \phi) \xrightarrow{x} (p, \phi) \;{}_A\|_B\; Q'} x \in B_\tau \setminus A$$

$$\frac{A, B \in \mathbb{P}(\Sigma) \wedge (p, \phi) \xrightarrow{a} P' \wedge (q, \phi) \xrightarrow{a} Q'}{(p \;{}_{e_1}\|_{e_2}\; q, \phi) \xrightarrow{a} P' \;{}_A\|_B\; Q'} a \in (A \cap B)_\surd$$

and the promoted operator

$$\frac{P \xrightarrow{x} P'}{P \ {}_A\|_B \ Q \xrightarrow{x} P' \ {}_A\|_B \ Q} x \in A_\tau \setminus B$$

$$\frac{Q \xrightarrow{x} Q'}{P \ {}_A\|_B \ Q \xrightarrow{x} P \ {}_A\|_B \ Q'} x \in B_\tau \setminus A$$

$$\frac{P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q'}{P \ {}_A\|_B \ Q \xrightarrow{a} P' \ {}_A\|_B \ Q'} a \in (A \cap B)_\surd$$

4.8 Hiding

For hiding we take

$$A \ = \ \varsigma(\mathcal{C}[\![e]\!]\phi)$$

then, provided A is finite,

$$\frac{A \notin \mathbb{P}(\Sigma)}{(p \setminus e, \phi) \xrightarrow{\tau} error}$$

$$\frac{A \in \mathbb{P}(\Sigma) \wedge (p, \phi) \xrightarrow{a} P'}{(p \setminus e, \phi) \xrightarrow{\tau} P' \setminus A} a \in A \qquad \frac{A \in \mathbb{P}(\Sigma) \wedge (p, \phi) \xrightarrow{x} P'}{(p \setminus e, \phi) \xrightarrow{x} P' \setminus A} x \notin A$$

and

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} a \in A \qquad \frac{P \xrightarrow{x} P'}{P \setminus A \xrightarrow{x} P' \setminus A} x \notin A$$

4.9 Conditionals

Conditionals evaluate the guard and branch to the appropriate alternative

$$\frac{\mathcal{C}[\![b]\!]\phi \notin B}{(\text{if } b \text{ then } p \text{ else } q, \phi) \xrightarrow{\tau} error}$$

$$\frac{\mathcal{C}[\![b]\!]\phi = True \wedge (p, \phi) \xrightarrow{x} P'}{(\text{if } b \text{ then } p \text{ else } q, \phi) \xrightarrow{x} P'} \qquad \frac{\mathcal{C}[\![b]\!]\phi = False \wedge (q, \phi) \xrightarrow{x} Q'}{(\text{if } b \text{ then } p \text{ else } q, \phi) \xrightarrow{x} Q'}$$

We do not give explicit clauses for the replicated operations; they are unrolled into the binary form of the operator, precisely as for the denotational semantics.

4.10 Prefix

Prefixing simply involves invoking the auxiliary functions defined for prefixing in the denotational semantics

$$\frac{\mathit{offer}[\![c\ fs]\!]\check{\phi} \notin \mathbb{P}(\Sigma)}{(c\ fs \rightarrow p, \phi) \xrightarrow{\tau} \mathit{error}}$$

$$\frac{a \in \mathit{offer}[\![c\ fs]\!]\check{\phi} \in \mathbb{P}(\Sigma) \land \rho = \mathit{after}(c\ fs, a, \check{\phi}) \land \phi' = \phi \oplus \hat{\rho}}{(c\ fs \rightarrow p, \phi) \xrightarrow{a} (p, \phi')}$$

where $\hat{\rho} = \rho \rhd E$ converts from $Env_{\mathcal{N}}$ to Env_C; we can ignore process valued components since we know that processes may not be communicated.

4.11 Renaming

Renaming builds the relation and then uses this to change the events. As usual, we must check the result of evaluating the relation before it can be used. If

$$R \quad = \quad \varsigma(\mathcal{C}[\![\{ss \mid gs\}]\!]\phi)$$

then

$$\frac{P \xrightarrow{x} P' \land x \simeq_R y}{P[\![R]\!] \xrightarrow{y} P'[\![R]\!]}$$

and

$$\frac{R \notin \mathbb{P}(\Sigma \times \Sigma)}{(p[\![ss \mid gs]\!], \phi) \xrightarrow{\tau} \mathit{error}}$$

$$\frac{(p, \phi) \xrightarrow{x} P' \land R = \mathcal{C}[\![\{ss \mid gs\}]\!]\phi \in \mathbb{P}(\Sigma \times \Sigma) \land x \simeq_R y}{(p[\![ss \mid gs]\!], \phi) \xrightarrow{y} P'[\![R]\!]}$$

where $\simeq_R$ is the same lifted version of the relation used earlier

$$x \simeq_R y \quad \Leftrightarrow \quad (x, y) \in R \lor (x = y \land x \notin \mathrm{dom}\, R)$$

The need to preserve finite non-determinism again requires that

$$\forall\, x \in \Sigma \bullet R^{-1}(x) \in \mathbb{F}(\Sigma)$$

4.12 Chaos

When defining $CHAOS$ we exploit the equivalence

$$CHAOS(A) \quad = \quad STOP \sqcap SKIP \sqcap \Box\, a : A \bullet a \rightarrow CHAOS(A)$$

If

$$A = \varsigma(\mathcal{C}[\![e]\!]\phi)$$

then

$$\frac{A \notin \mathbb{P}(\Sigma)}{(CHAOS(e), \phi) \xrightarrow{\tau} error}$$

$$\frac{A \in \mathbb{P}(\Sigma)}{(CHAOS(e), \phi) \xrightarrow{\tau} STOP} \qquad \frac{A \in \mathbb{P}(\Sigma)}{(CHAOS(e), \phi) \xrightarrow{\checkmark} STOP}$$

$$\frac{A \in \mathbb{P}(\Sigma)}{(CHAOS(e), \phi) \xrightarrow{a} CHAOS(A)} a \in A$$

and

$$\frac{}{CHAOS(A) \xrightarrow{\tau} STOP} \qquad \frac{}{CHAOS(A) \xrightarrow{\checkmark} STOP}$$

$$\frac{}{CHAOS(A) \xrightarrow{a} CHAOS(A)} a \in A$$

4.13 Bindings

When looking up names in the environment, we prefix the expansion with an internal action so that unguarded recursions will produce an infinite sequence of internal actions.

$$\frac{\phi(n) \in C}{(n, \phi) \xrightarrow{\tau} \phi(n)} \qquad \frac{\phi(n) \notin C}{(n, \phi) \xrightarrow{\tau} error}$$

Names are bound using an adaptation of the machinery from the denotational semantics

$$\frac{\phi' = \phi \oplus \bigcup \mathcal{B}_C[\![d_i]\!]\phi' \wedge (p, \phi') \xrightarrow{x} P'}{(\text{let } d_1, \ldots d_n \text{ within } p, \phi) \xrightarrow{x} P'}$$

where $\mathcal{B}_C$ corresponds to $\mathcal{B}_{\mathcal{N}}$.

Applications can be unrolled, providing the result is a process

$$\frac{\mathcal{C}[\![p]\!]\phi \in E \to C}{(p(es), \phi) \xrightarrow{\tau} apply(\mathcal{C}[\![p]\!]\phi, \mathcal{C}[\![(es)]\!]\phi)} \qquad \frac{\mathcal{C}[\![p]\!]\phi \notin E \to C}{(p(es), \phi) \xrightarrow{\tau} error}$$

4.14 Summary

This chapter has identified the configurations in the enriched language which correspond to the closed-terms used in the operational semantics of [7]. A transition system has been defined over those configurations, giving an operational semantics for the language presented here.

The next chapter lays the foundations for showing that the operational semantics is congruent to the denotational semantics.

Chapter 5

Tree semantics

We now build a semantic model where the failures-divergences component of the denotational semantics is replaced by synchronisation trees. These record the same transitions as the operational semantics, but also provide a fixed point theory similar to that used in the denotational semantics. This allows them to act as a bridge between those two semantics in the subsequent congruence proof.

The basic approach taken is that used in the (unpublished) [7], but the semantics in that paper dealt only with process terms, showing a congruence between $\mathcal{N}$ and an operational semantics defined over closed process terms. This chapter extends this to cover the expression component of CSP in the same way that Chapter 3 extended the denotational semantics.

A complete proof of the congruence will be presented in this chapter and the next, but some of the technical details concerning synchronisation trees (which are taken directly from [7]) are deferred until Appendix A.

5.1 Trees

A synchronisation tree is expressible as the sum of a set of trees, each prefixed by an event so that for $t \in T$ it may be expressed, uniquely subject to reordering, as

$$t \quad = \quad \Sigma_{i \in I} x_i . t_i$$

where $x_i \in \Sigma_\tau$ and $t_i \in T$ and the finite-branching property holds, in that

$$\forall\, x \in \Sigma_\tau \bullet \{ i \in I \mid x = x_i \} \in \mathbb{F}(I)$$

We write the empty tree as *nil* so that

$$nil \quad = \quad \Sigma_{i \in \{\}}$$

We also abbreviate the cases of summing a singleton or pair

$$\begin{aligned} x.t &= \Sigma\{x.t\} \\ x_1.t_1 + x_2.t_2 &= \Sigma\{x_1.t_1, x_2.t_2\} \end{aligned}$$

(Note that Σ is used for two separate purposes; the set of events and the sum operation on trees. These may be distinguished by context.)

There exist operations on synchronisation trees

$$\begin{aligned} _ \to _ &: \mathbb{P}(\Sigma) \times (\Sigma \to T) \to T \\ _\,;_, _ \triangle _, _ \,|||\, _, _ \underset{A}{\|} _, _ \,{}_A\|_B\, _, _ \,\Box\, _, _ \sqcap _ &: T \times T \to T \\ _ \setminus A, _[\![R]\!] &: T \to T \\ CHAOS(_) &: \mathbb{F}(\Sigma) \to T \end{aligned}$$

given by

$$\begin{aligned} A \to f &= \Sigma\{a.f(a) \mid a \in A\} \\ s\,;t &= \Sigma_{x_i \neq \checkmark} x_i.(s_i\,;t) + \Sigma_{x_i = \checkmark} \tau.t \\ s \triangle t &= \Sigma x_i.(s_i \triangle t) + \Sigma_{y_i = \tau} y_i.(s \triangle t_i) + \Sigma_{y_i \neq \tau} y_i.t_i \\ s \,|||\, t &= \left\{ \begin{array}{l} \Sigma_{x_i \neq \checkmark} x_i.(s_i \,|||\, t) \\ + \\ \Sigma_{y_i \neq \checkmark} y_i.(s \,|||\, t_i) \\ + \\ \Sigma_{x_i = y_j = \checkmark} \checkmark.(s_i \,|||\, t_j) \end{array} \right. \\ s \underset{A}{\|} t &= \left\{ \begin{array}{l} \Sigma_{x_i \notin A_\checkmark} x_i.(s_i \underset{A}{\|} t) \\ + \\ \Sigma_{y_i \notin A_\checkmark} y_i.(s \underset{A}{\|} t_i) \\ + \\ \Sigma_{x_i = y_j \in A_\checkmark} x_i.(s_i \underset{A}{\|} t_j) \end{array} \right. \\ s \,{}_A\|_B\, t &= \left\{ \begin{array}{l} \Sigma_{x_i \in A \setminus B} x_i.(s_i \,{}_A\|_B\, t) \\ + \\ \Sigma_{y_i \in B \setminus A} y_i.(s \,{}_A\|_B\, t_i) \\ + \\ \Sigma_{x_i = y_j \in (A \cap B)_\checkmark} x_i.(s_i \underset{A}{\|} t_j) \end{array} \right. \\ s \,\Box\, t &= \Sigma_{x_i \neq \tau} x_i.s_i + \Sigma_{y_i \neq \tau} y_i.t_i + \Sigma_{x_i = \tau} \tau.(s_i \,\Box\, t) + \Sigma_{y_i = \tau} \tau.(s \,\Box\, t_i) \\ s \sqcap t &= \tau.s + \tau.t \\ s \setminus A &= \Sigma_{x_i \notin A} x_i.(s_i \setminus A) + \Sigma_{x_i \in A} \tau.s_i \\ s[\![R]\!] &= \Sigma\{x.(s_i[\![R]\!]) \mid x \in \Sigma_\tau \wedge ((x_i, x) \in R \vee x_i = x \notin \operatorname{dom} R)\} \\ CHAOS(A) &= \tau.STOP + \Sigma_{x \in A_\checkmark} x.CHAOS(A) \end{aligned}$$

where $s = \Sigma x_i.s_i$ and $t = \Sigma y_i.t_i$.

The model presented for synchronisation trees in [7] is such that T has a metric defined on it in terms of a cut operation.

$$\forall\, x \in T, m \in \mathbb{N}, n \in \mathbb{N} \bullet (x \downarrow m) \downarrow n = x \downarrow (m \sqcap n)$$
$$\forall\, x, y \in T \bullet x \downarrow 0 = y \downarrow 0$$
$$\forall\, x, y \in T \bullet x \neq y \Rightarrow \exists\, n \in \mathbb{N} \bullet x \downarrow n \neq y \downarrow n$$

The operators defined above have properties which ensure that equations defined using the operations have unique fixed points under the metric. A summary of the required results can be found in Appendix A.

5.2 Semantics

We define $\mathcal{S} : Any \to Env_T \to D_T$. For non-process elements this is exactly the same as for $\mathcal{E} : Any \to Env_{\mathcal{N}} \to D_{\mathcal{N}}$, and we omit the details. It suffices to provide definitions for the process productions in the syntax.

The atomic processes are simple

$$\begin{aligned} \mathcal{S}[\![STOP]\!]\psi &= nil \\ \mathcal{S}[\![SKIP]\!]\psi &= \checkmark.nil \end{aligned}$$

as are the operations which do not involve expression components

$$\begin{aligned} \mathcal{S}[\![p\ ;\ q]\!]\psi &= \mathcal{S}[\![p]\!]\psi\ ;\ \mathcal{S}[\![q]\!]\psi \\ \mathcal{S}[\![p\ \triangle\ q]\!]\psi &= \mathcal{S}[\![p]\!]\psi\ \triangle\ \mathcal{S}[\![q]\!]\psi \\ \mathcal{S}[\![p\ |||\ q]\!]\psi &= \mathcal{S}[\![p]\!]\psi\ |||\ \mathcal{S}[\![q]\!]\psi \\ \mathcal{S}[\![p \,\Box\, q]\!]\psi &= \mathcal{S}[\![p]\!]\psi \,\Box\, \mathcal{S}[\![q]\!]\psi \\ \mathcal{S}[\![p \sqcap q]\!]\psi &= \mathcal{S}[\![p]\!]\psi \sqcap \mathcal{S}[\![q]\!]\psi \end{aligned}$$

Prefixing needs to build both the set of events offered and the function which yields the resulting process, given an event from this set.

$$\mathcal{S}[\![c\ fs \to P]\!]\psi \quad = \quad offer[\![c\ fs]\!]\psi \to (\lambda\, e \bullet \mathcal{S}[\![P]\!](after(c\ fs, e, \psi))$$

where *offer* and *after* are similar to the versions used in the definition of $\mathcal{E}$, operating on Env_T rather than $Env_{\mathcal{N}}$.

As with the denotational semantics in Chapter 3, if $\perp$ is produced as part of the expression evaluations needed to evaluate a process term, then the process term should evaluate to the worst process; here that corresponds to τ^*.

Hiding is comparatively simple

$$\mathcal{S}[\![p \setminus e]\!]\psi \;=\; \mathcal{S}[\![p]\!]\psi \setminus A$$

where $A = \varsigma(\mathcal{S}[\![e]\!]\psi)$ and provided $A \in \mathbb{F}(\Sigma)$.

For sharing we have

$$\mathcal{S}[\![p \underset{e}{\|} q]\!]\psi \;=\; \mathcal{S}[\![p]\!]\psi \underset{A}{\|} \mathcal{S}[\![q]\!]\psi$$

where $A = \varsigma(\mathcal{S}[\![e]\!]\psi)$ and provided $A \in \mathbb{P}(\Sigma)$.

For parallel we have

$$\mathcal{S}[\![p \;{}_e\|_{e'}\; q]\!]\psi \;=\; \mathcal{S}[\![p]\!]\psi \;{}_A\|_B\; \mathcal{S}[\![q]\!]\psi$$

where $A = \varsigma(\mathcal{S}[\![e]\!]\psi)$, $B = \varsigma(\mathcal{S}[\![e']\!]\psi)$ and provided $A, B \in \mathbb{P}(\Sigma)$.

For relabelling we build and use the relation

$$\mathcal{S}[\![p\,[\![ss \mid gs]\!]\,]\!]\psi \;=\; (\mathcal{S}[\![p]\!]\psi)\,[\![R]\!]$$

where $R = \varsigma(\mathcal{S}[\![\{ss \mid gs\}]\!]\psi)$ and provided $R \in \mathbb{P}(\Sigma \times \Sigma)$.

For conditionals

$$\mathcal{S}[\![\text{if } b \text{ then } p \text{ else } q]\!]\psi \;=\; \begin{cases} \mathcal{S}[\![p]\!]\psi & \text{, if } x = \mathit{True} \\ \mathcal{S}[\![q]\!]\psi & \text{, if } x = \mathit{False} \\ \tau^* & \text{, if } x = \bot \end{cases}$$

where $x = \mathcal{S}[\![b]\!]\psi$ and provided $x \in B_\bot$.

For chaos we have

$$\mathcal{S}[\![CHAOS(e)]\!]\psi \;=\; CHAOS(A)$$

where $A = \varsigma(\mathcal{S}[\![e]\!]\psi)$ and provided $A \in \mathbb{F}(\Sigma)$.

The replicated operators are unrolled, precisely as in the definition of $\mathcal{E}$, so details are again omitted.

Local definitions are unrolled precisely as in the other semantic models; the only difference is the semantic function used used to create the values, so that

$$\begin{aligned} \mathcal{S}[\![\text{let } ds \text{ within } e]\!]\psi &= \mathcal{S}[\![e]\!]\psi' \\ \psi' &= \psi \oplus \bigcup \mathcal{B}_T[\![d_i]\!]\psi' \end{aligned}$$

where

$$\mathcal{B}_T[\![x = e]\!]\rho \;=\; bind(x, \mathcal{S}[\![e]\!]\psi)$$

The fixed point operation required here is complicated by the fact that D_T has a metric space T embedded in a domain. However, the fact that non-process values depend only upon non-process values means that it is possible to form the fixed point in two stages. The first stage giving fixed points to the non-process terms using the domain, and the second using the metric space to provide fixed points for the process values.

5.3 Connection to the operational semantics

The operational semantics of Chapter 4 is defined in terms of the transition relation $\rightarrow \subseteq \Gamma \times \Sigma_\tau \times \Gamma$. Hence, for $P \in \Gamma$ the relation defines $(P \rightarrow) \subseteq \Sigma_\tau \times \Gamma$. Synchronisation trees also have the form $t = \Sigma_{i \in I} x_i.t_i$ where $\{x_i\} \subseteq \Sigma_\tau$. This suggest that there is a natural mapping from the operational semantics defined by the transition relation to synchronisation trees. This section defines that mapping and demonstrates that it respects the process operators.

Consider $T^+ = T \cup \{error\}$, then there is a function $\mathcal{U} : \Gamma \rightarrow T^+$ defined by

$$\begin{aligned} \mathcal{U}(error) &= error \\ \mathcal{U}(P) &= \begin{cases} \tau^* & \text{, if } P \in \Gamma_\perp \\ \Sigma\{x.\mathcal{U}(Q) \mid P \xrightarrow{x} Q\} & \text{, if } \nexists x, Q \bullet P \xrightarrow{x} Q \wedge \mathcal{U}(Q) = error \\ error & \text{, otherwise} \end{cases} \end{aligned}$$

Essentially, transitions are mapped across directly from those of the transition system to those of the synchronisation tree. The only exception to this is that any state of the transition system which leads to the error configuration in Γ is mapped to the error value in T^+.

We can lift $\mathcal{U}$ to a function from D_C to D_T by defining

$$\widehat{\mathcal{U}}(x) = \begin{cases} x & \text{, if } x \in E_\perp \\ \mathcal{U}(x) & \text{, if } x \in C \\ \mathcal{U} \circ x & \text{, if } x \in E \rightarrow C \end{cases}$$

We now show that the CSP operators defined in the operational semantics correspond to the tree operators under $\mathcal{U}$.

Theorem 1 *For $P, Q \in \Gamma$ such that $\mathcal{U}(P) \neq error$ and $\mathcal{U}(Q) \neq error$*

$$\begin{aligned} \mathcal{U}(P \,\Box\, Q) &= \mathcal{U}(P) \,\Box\, \mathcal{U}(Q) \\ \mathcal{U}(P \sqcap Q) &= \mathcal{U}(P) \sqcap \mathcal{U}(Q) \\ \mathcal{U}(P \,;\, Q) &= \mathcal{U}(P) \,;\, \mathcal{U}(Q) \end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{U}(P \triangle Q) & = & \mathcal{U}(P) \triangle \mathcal{U}(Q) \\
\mathcal{U}(P \,|||\, Q) & = & \mathcal{U}(P) \,|||\, \mathcal{U}(Q) \\
\mathcal{U}(P \underset{A}{\parallel} Q) & = & \mathcal{U}(P) \underset{A}{\parallel} \mathcal{U}(Q) \\
\mathcal{U}(P \,{}_A\|_B\, Q) & = & \mathcal{U}(P) \,{}_A\|_B\, \mathcal{U}(Q) \\
\mathcal{U}(P \setminus A) & = & \mathcal{U}(P) \setminus A \\
\mathcal{U}(P[\![R]\!]) & = & \mathcal{U}(P)[\![R]\!] \\
\mathcal{U}(CHAOS(A)) & = & CHAOS(A)
\end{array}
$$

The result fails to hold in the event of error values, since the right-hand side is strict in error, while the left-hand side may fail to reach the error configuration. The expressions (such as $P \sqcap Q$) on the left-hand side are taken to include the non-promoted form of the configurations (such as $(p \sqcap q, \phi)$).

Proof

The proof is by induction on n that

$$
\begin{array}{rcl}
\mathcal{U}((p \oplus q, \phi)) \downarrow n & = & (\mathcal{U}((p,\phi)) \oplus \mathcal{U}((q,\phi))) \downarrow n \\
\mathcal{U}(P \oplus Q) \downarrow n & = & (\mathcal{U}(P) \oplus \mathcal{U}(Q)) \downarrow n \\
\mathcal{U}((\oplus p, \phi)) \downarrow n & = & (\oplus \mathcal{U}((p,\phi))) \downarrow n
\end{array}
$$

The case $\oplus = \underset{A}{\parallel}$ is shown here; the other cases are similar. The case $n = 0$ is of course trivial since $x \downarrow 0 = nil$.

Suppose $\mathcal{U}(P) = \Sigma x_i.s_i$ and $\mathcal{U}(Q) = \Sigma y_i.t_i$, then assuming the result true for n

$$
\begin{array}{l}
\mathcal{U}(P \underset{A}{\parallel} Q) \downarrow (n{+}1) \\
\quad = \quad \Sigma\{x.(\mathcal{U}(R) \downarrow n) \mid P \underset{A}{\parallel} Q \xrightarrow{x} R\} \\
\quad = \quad \left\{ \begin{array}{l}
\Sigma\{x.(\mathcal{U}(P' \underset{A}{\parallel} Q) \downarrow n) \mid x \notin A_\surd \wedge P \xrightarrow{x} P'\}+ \\
\Sigma\{x.(\mathcal{U}(P \underset{A}{\parallel} Q') \downarrow n) \mid x \notin A_\surd \wedge Q \xrightarrow{x} Q'\}+ \\
\Sigma\{x.(\mathcal{U}(P' \underset{A}{\parallel} Q') \downarrow n) \mid x \in A_\surd \wedge P \xrightarrow{x} P' \wedge Q \xrightarrow{x} Q'\}
\end{array} \right. \\
\quad = \quad \left\{ \begin{array}{l}
\Sigma\{x.((\mathcal{U}(P') \underset{A}{\parallel} \mathcal{U}(Q)) \downarrow n) \mid x \notin A_\surd \wedge P \xrightarrow{x} P'\}+ \\
\Sigma\{x.((\mathcal{U}(P) \underset{A}{\parallel} \mathcal{U}(Q')) \downarrow n) \mid x \notin A_\surd \wedge Q \xrightarrow{x} Q'\}+ \\
\Sigma\{x.((\mathcal{U}(P') \underset{A}{\parallel} \mathcal{U}(Q')) \downarrow n) \mid x \in A_\surd \wedge P \xrightarrow{x} P' \wedge Q \xrightarrow{x} Q'\}
\end{array} \right. \\
\quad = \quad \left\{ \begin{array}{l}
(\Sigma\{x_i.(s_i \underset{A}{\parallel} \mathcal{U}(Q)) \mid x_i \notin A_\surd\}+ \\
\Sigma\{y_j.(\mathcal{U}(P) \underset{A}{\parallel} t_j) \mid y_j \notin A_\surd\}+ \\
\Sigma\{x_i.(s_i \underset{A}{\parallel} t_j) \mid x_i = y_j \in A_\surd\}) \downarrow (n{+}1)
\end{array} \right.
\end{array}
$$

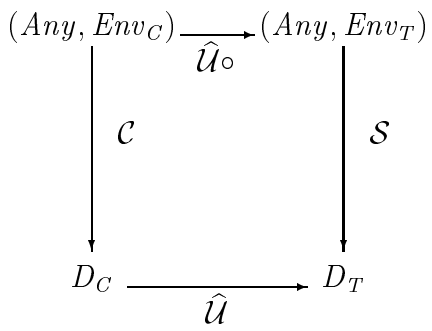


Figure 5.1: Congruence of operational and tree semantics

$$
\begin{aligned}
&= \left\{ \begin{array}{l} (\Sigma_{x_i \notin A_\surd} x_i.(s_i \underset{A}{\|} \mathcal{U}(Q)) + \\ \Sigma_{y_j \notin A_\surd} y_j.(\mathcal{U}(P) \underset{A}{\|} t_j) + \\ \Sigma_{x_i = y_j \in A_\surd} x_i.(s_i \underset{A}{\|} t_j)) \downarrow (n+1) \end{array} \right. \\
&= (\mathcal{U}(P) \underset{A}{\|} \mathcal{U}(Q)) \downarrow (n+1)
\end{aligned}
$$

proving the inductive step. □

Given this result for the operators, we next lift this result to cover general elements of $Any \times Env_C$.

Theorem 2 *For all $e \in Any$ and $\phi \in Env_C$,*

$$\widehat{\mathcal{U}}(\mathcal{C}[\![e]\!]\phi) \;=\; \mathcal{S}[\![e]\!](\widehat{\mathcal{U}} \circ \phi)$$

provided $\mathcal{S}[\![e]\!](\widehat{\mathcal{U}} \circ \phi) \neq error$. This implies that the diagram in figure 5.1 commutes.

Proof

This is proved by structural induction on e. If the value is not a process, the result follows from the fact that the defining clauses of $\mathcal{C}$ and $\mathcal{S}$ for non-process values are identical (and are those of $\mathcal{E}$.)

For processes, the result follows from 2. For example

$$
\begin{aligned}
\widehat{\mathcal{U}}(\mathcal{C}[\![p \,\Box\, q]\!]\phi) &= \widehat{\mathcal{U}}(\mathcal{C}[\![p]\!]\phi \,\Box\, \mathcal{C}[\![q]\!]\phi) \\
&= \widehat{\mathcal{U}}(\mathcal{C}[\![p]\!]\phi) \,\Box\, \widehat{\mathcal{U}}(\mathcal{C}[\![q]\!]\phi) \\
&= \mathcal{S}[\![p]\!](\widehat{\mathcal{U}} \circ \phi) \,\Box\, \mathcal{S}[\![q]\!](\widehat{\mathcal{U}} \circ \phi) \\
&= \mathcal{S}[\![p \,\Box\, q]\!](\widehat{\mathcal{U}} \circ \phi)
\end{aligned}
$$

For recursive definitions, we have

$$\widehat{\mathcal{U}}(\mathcal{C}[\![\text{let } ds \text{ within } e]\!]\phi) \;=\; \widehat{\mathcal{U}}(\mathcal{C}[\![e]\!]\phi')$$

where

$$\phi' \;=\; \phi \oplus \bigcup \mathcal{B}_C[\![d_i]\!]\phi'$$

so that

$$\begin{aligned}\widehat{\mathcal{U}} \circ \phi' &= (\widehat{\mathcal{U}} \circ \phi) \oplus \bigcup\{n_i \mapsto \widehat{\mathcal{U}}(\mathcal{C}[\![x_i]\!]\phi')\} \\ &= (\widehat{\mathcal{U}} \circ \phi) \oplus \bigcup\{n_i \mapsto \mathcal{S}[\![x_i]\!](\widehat{\mathcal{U}} \circ \phi')\} \\ &= (\widehat{\mathcal{U}} \circ \phi) \oplus \bigcup \mathcal{B}_T[\![d_i]\!](\widehat{\mathcal{U}} \circ \phi')\end{aligned}$$

which is the defining equation for ψ' where

$$\mathcal{S}[\![\text{let } ds \text{ within } e]\!](\widehat{\mathcal{U}} \circ \phi) \;=\; \mathcal{S}[\![e]\!]\psi'$$

From this it follows that

$$\widehat{\mathcal{U}}(\mathcal{C}[\![\text{let } ds \text{ within } e]\!]\phi) \;=\; \mathcal{S}[\![\text{let } ds \text{ within } e]\!](\widehat{\mathcal{U}} \circ \phi)$$

since the fixed point calculation either lies entirely within in E (where both $\mathcal{C}$ and $\mathcal{S}$ revert to $\mathcal{E}$) or within T or $E \to T$ in which case the fix-point is calculated in terms of a contraction mapping and so is unique. In both cases, the fixed point is identical on both sides. □

5.4 Summary

We now have an intermediate semantic space, T, and a semantic function, $\mathcal{S}$, defined using this space. We have also shown that this intermediate semantics is congruent to the operational semantics of Chapter 4 under the function $\mathcal{U}$.

The next chapter will connect this intermediate semantics to the denotational semantics of Chapter 3.

Chapter 6

Congruence

We now show that the denotational model is equivalent to the operational semantics. The first stage is to describe a general method for moving from a transition system to the failures-divergences model.

6.1 Mapping to processes

A transition system is taken to be a triple $(S, \Theta, \rightarrow)$ where S is the set of states, Θ is the set of actions and $\rightarrow$ is the transition relation, so that

$$\rightarrow \subseteq S \times \Theta \times S$$

In the cases considered here, Θ will be based on the event-set Σ with the addition of a (finite) number of internal actions so that

$$\Sigma \subseteq \Theta$$
$$\Theta \setminus \Sigma \text{ is finite}$$

We also consider a distinguished state $error \in S$ which as usual represents a type-incorrect input. This state will always be final, so that

$$\nexists x \in \Theta, s \in S \bullet error \xrightarrow{x} s$$

There is a natural extension of the transition relation to traces over Θ, so that

$$P \xrightarrow{\langle\rangle} Q \Leftrightarrow P = Q$$
$$P \xrightarrow{\langle x \rangle} Q \Leftrightarrow P \xrightarrow{x} Q$$
$$P \xrightarrow{s \frown t} Q \Leftrightarrow \exists R \in S \bullet P \xrightarrow{s} R \xrightarrow{t} Q$$

for $P, Q \in S$ and then to traces over Σ

$$P \stackrel{s}{\Longrightarrow} Q \;\Leftrightarrow\; \exists\, t \in \Theta^* \bullet s = t \upharpoonright \Sigma \wedge P \stackrel{t}{\longrightarrow} Q$$

In terms of these we define, for $P \in S$ and $X \subseteq \Sigma$

$$\begin{array}{rcl} P \ ref\ X & \Leftrightarrow & \forall\, x \in X \cup (\Theta \setminus \Sigma) \bullet \nexists\, Q \in S \bullet P \stackrel{x}{\longrightarrow} Q \\ stable(P) & \Leftrightarrow & \nexists\, Q \in S \bullet P \stackrel{\cdot}{\longrightarrow} Q \\ P \uparrow & \Leftrightarrow & \exists \langle P_i \mid i \in \mathbb{N} \rangle \bullet P_0 = P \wedge \forall\, i \in \mathbb{N} \bullet P_i \stackrel{\cdot}{\longrightarrow} P_{i+1} \end{array}$$

where

$$P \stackrel{\cdot}{\longrightarrow} Q \;\Leftrightarrow\; \exists\, x \in \Theta \setminus \Sigma \bullet P \stackrel{x}{\longrightarrow} Q$$

Given these, $P_S : S \to \mathbb{P}(\Sigma^* \times \mathbb{P}(\Sigma)) \times \mathbb{P}(\Sigma^*) \cup \{error\}$ can be defined as

$$P_S(P) \;=\; \begin{cases} error & , \text{if } \exists\, s \in \Theta^* \bullet P \stackrel{s}{\longrightarrow} error \\ (\mathcal{F}_S(P), \mathcal{D}_S(P)) & , \text{otherwise} \end{cases}$$

where

$$\begin{array}{rcl} \mathcal{F}_S(P) & = & \left\{ \begin{array}{l} \{(s, X) \mid \exists\, Q \in S \bullet P \stackrel{s}{\Longrightarrow} Q \wedge Q \ ref\ X\} \\ \cup \\ \{(s, X) \mid \exists\, Q \in S \bullet P \stackrel{s}{\Longrightarrow} Q \wedge Q \uparrow\} \end{array} \right. \\ \mathcal{D}_S(P) & = & \{s^\frown t \mid \exists\, Q \in S \bullet P \stackrel{s}{\Longrightarrow} Q \wedge Q \uparrow\} \end{array}$$

We must now consider when the values given by P_S lie in $\mathcal{N} \cup \{error\}$. We begin with some useful results about the relations used to define P_S.

Lemma 1 $\forall\, P \in S \bullet P \uparrow \vee\, \exists\, Q \in S \bullet stable(Q) \wedge P \stackrel{\langle\rangle}{\Longrightarrow} Q$

Proof

From the definition of $P \uparrow$, either we can generate a sequence of processes $\langle P_i \mid i \in \mathbb{N} \rangle$ or we fail to do so, since we reach a stable process. □

Lemma 2 $P \stackrel{s^\frown t}{\Longrightarrow} Q \Rightarrow \exists\, R \bullet (stable(R) \vee R \uparrow) \wedge P \stackrel{s}{\Longrightarrow} R$

Proof

From the definition of $\stackrel{s^\frown t}{\Longrightarrow}$, there exists R' such that $P \stackrel{s}{\Longrightarrow} R'$ and then by Lemma 1 there exists a stable or divergent R such that $R' \stackrel{\langle\rangle}{\Longrightarrow} R$ and so $P \stackrel{s}{\Longrightarrow} R$. □

Lemma 3 *If S is finite-branching, then for all $P \in S$ and $s \in \Sigma^*$,*

$$\{R \mid P \stackrel{s}{\Longrightarrow} R\} \text{ is finite}$$

or

$$\exists\, t \leqslant s, Q \in S \bullet P \stackrel{t}{\Longrightarrow} Q \wedge Q \uparrow$$

Proof

Consider the tree t associated with P where the branches at each node correspond to the (possibly repeated) transitions at that point. Restrict t to the subtree t' such that all paths through t' correspond to prefixes of s, then at each node, transitions are labelled either with the next visible action in s or with internal actions, so since $(S, \rightarrow)$ is finite-branching and $\Theta \setminus \Sigma$ is finite, t' is finite-branching.

If t' is finite then the set, which corresponds to the leaves of t', is also finite. If t' is infinite then by Koenig's Lemma it has an infinite path. Since s is finite, this infinite path must have a infinite number of (contiguous) internal actions. Hence there is a divergent point in the tree and the alternative holds. □

Definition 1 *Transition system $(S, \Theta, \rightarrow)$ is finite-branching if and only if for all $P \in S$ and $x \in \Theta$ the set $\{Q \in S \mid P \stackrel{x}{\longrightarrow} Q\}$ is finite.*

Theorem 3 *P_S yields processes from finite-branching transition systems.*

Proof

We must simply check that the following axioms for processes hold.

$$F \neq \{\} \tag{6.1}$$

$$(s^\frown t, \{\}) \in F \;\Rightarrow\; (s, \{\}) \in F \tag{6.2}$$

$$(t, X) \in F \wedge Y \subseteq X \;\Rightarrow\; (t, Y) \in F \tag{6.3}$$

$$(t, X) \in F \wedge (\forall\, a \in Y \bullet (t^\frown \langle a \rangle, \{\}) \notin F) \;\Rightarrow\; (t, X \cup Y) \in F \tag{6.4}$$

$$(\forall\, X' \in \mathbb{F}(X) \bullet (t, X') \in F) \;\Rightarrow\; (t, X) \in F \tag{6.5}$$

$$s \in D \;\Rightarrow\; s^\frown t \in D \tag{6.6}$$

$$s \in D \;\Rightarrow\; (s^\frown t, X) \in F \tag{6.7}$$

The first axiom follows directly from Lemma 1.

Axiom 2 is trivial, for assuming that

$$(s^\frown t, \{\}) \in \mathcal{F}(P)$$

we have that

$$\exists\, Q \bullet P \stackrel{s^\frown t}{\Longrightarrow} Q$$

and by Lemma 2

$$\exists\, R \bullet P \stackrel{s}{\Longrightarrow} R \wedge (\mathit{stable}(R) \vee R\uparrow)$$

so

$$(s, \{\}) \in \mathcal{F}(P)$$

Axiom 3 is even simpler, since it follows from

$$P \ \mathit{ref}\ X \wedge Y \subseteq X \Rightarrow P \ \mathit{ref}\ Y$$

For axiom 4

$$(s, X) \in \mathcal{F}(P) \wedge \forall\, a \in Y \bullet (s^\frown\langle a\rangle, \{\}) \notin \mathcal{F}(P)$$

implies

$$\exists\, Q \bullet P \stackrel{s}{\Longrightarrow} Q \wedge (Q \ \mathit{ref}\ X \vee Q\uparrow) \wedge \forall\, a \in Y \bullet \nexists R \bullet P \stackrel{s^\frown\langle a\rangle}{\Longrightarrow} R$$

which implies

$$\exists\, Q \bullet P \stackrel{s}{\Longrightarrow} Q \wedge (Q\uparrow \vee \forall\, a \in X \cup (\Theta \setminus \Sigma) \cup Y \bullet \nexists R \bullet Q \stackrel{a}{\longrightarrow} R)$$

which is just

$$(s, X \cup Y) \in \mathcal{F}(P)$$

For axiom 5, assume

$$\forall\, X' \in \mathbb{F}(X) \bullet (s, X') \in \mathcal{F}(P)$$

then by Lemma 3

$$\{R \mid P \stackrel{s}{\Longrightarrow} R\}\mathit{finite} \vee \exists\, t \leqslant s, Q \in S \bullet P \stackrel{t}{\Longrightarrow} Q \wedge Q\uparrow$$

The latter of these implies that $(s, X) \in \mathcal{F}(P)$ by the treatment of divergence in the definition of $\mathcal{F}$. Thus, suppose that the first holds and the second does not and take

$$S = \{\, Q \mid P \stackrel{s}{\Longrightarrow} Q \wedge \mathit{stable}(Q)\}$$

By assumption, S is finite and by Lemma 1 it is non-empty. Assuming

$$\nexists Q \in S \bullet Q \; ref \; X$$

we have that

$$\forall\, Q \in S \bullet \exists\, a_Q \in X, R \in S \bullet Q \xrightarrow{a_Q} R$$

and so

$$(s, \{a_Q \mid Q \in S\}) \notin \mathcal{F}(P)$$

but we also have that $\{a_Q \mid Q \in S\} \in \mathbb{F}(X)$ so

$$(s, \{a_Q \mid Q \in S\}) \in \mathcal{F}(P)$$

which is a contradiction. Thus

$$\exists\, Q \in S \bullet Q \; ref \; X$$

and so

$$(s, X) \in \mathcal{F}(P)$$

as required.

Axioms 6 and 7

$$s \in D \;\Rightarrow\; s^\frown t \in D$$
$$s \in D \;\Rightarrow\; (s^\frown t, X) \in F$$

follow trivially from the definition of $\mathcal{D}$. □

Lemma 4 *$(\Gamma, \Sigma_\tau, \rightarrow)$ is a finite-branching transition system.*

Proof

This follows from a simple induction based on the transition rules and the fact that the replicated operations are finite. □

Lemma 5 *T is a finite-branching transition system where the transition relation is given by*

$$(\Sigma x_i.t_i) \xrightarrow{x} t \;\Leftrightarrow\; \exists\, i \bullet x = x_i \wedge t = t_i$$

Proof

Trivially true, since T was defined to satisfy the finite-branching property. □

Theorem 4 *Both*

$$P_\Gamma : \Gamma \to \mathcal{N} \cup \{error\}$$

and

$$P_T : T^+ \to \mathcal{N} \cup \{error\}$$

are well-defined.

Proof

Follows directly from Lemma 4, Lemma 5 and Theorem 1. □

We now show how these two functions are related.

Theorem 5 $P_\Gamma = P_T \circ \mathcal{U}$

Proof

From the definition of $\mathcal{U}$ and the transition relation over T,

$$\mathcal{U}(P) \xrightarrow{x} \mathcal{U}(Q) \Leftrightarrow P \xrightarrow{x} Q$$

As P_S is determined entirely by the transition relation, the result follows directly. □

Note that both these functions can be lifted in the same way as $\mathcal{U}$ to yield

$$\begin{array}{lcl} \widehat{P_T} & : & D_T \to D_\mathcal{N} \\ \widehat{P_\Gamma} & : & D_C \to D_\mathcal{N} \end{array}$$

6.2 Review

So far we have defined three separate semantic models, which are related by the functions shown in Figure 6.1.

The basic starting point is a piece of syntax together with a list of definitions to provide a context for that syntax. The context is converted into an environment using the appropriate binding function. The environments can then be converted by composing a conversion function (either $\widehat{\mathcal{U}}$ or $\widehat{P_T}$) with them. Given a piece of syntax and an environment, the semantic functions then map this into the semantic domains which are again related by $\widehat{\mathcal{U}}$ and $\widehat{P_T}$. Note that in the case of D_C the bulk of the evaluation for processes is carried out by $\mathcal{U}$ using the transition relation; the semantic function $\mathcal{C}$ yields simple closures from process terms.

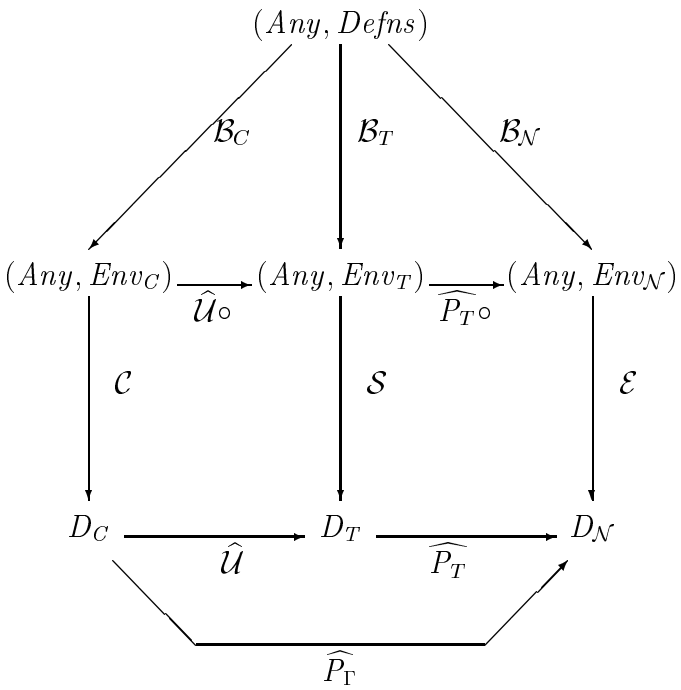


Figure 6.1: Relationship between semantic models

From Theorem 2 we know that the left-hand side of the diagram commutes, and Theorem 5 provides the same result for the lower part of the diagram.

Next we show that the tree operations in T correspond to the same operations in $\mathcal{N}$ under P_T and then we use this result to show that the right-hand side of the diagram also commutes. This will suffice to show that the diagram commutes as a whole, thus connecting the operational and denotational semantics.

Theorem 6 *For $s, t \in T$*

$$\begin{aligned}
P_T(s \,\Box\, t) &= P_T(s) \,\Box\, P_T(t) \\
P_T(s \sqcap t) &= P_T(s) \sqcap P_T(t) \\
P_T(s \,;\, t) &= P_T(s) \,;\, P_T(t) \\
P_T(s \,\triangle\, t) &= P_T(s) \,\triangle\, P_T(t) \\
P_T(s \,|||\, t) &= P_T(s) \,|||\, P_T(t) \\
P_T(s \underset{A}{\|} t) &= P_T(s) \underset{A}{\|} P_T(t) \\
P_T(s \;{}_A\|_B\; t) &= P_T(s) \;{}_A\|_B\; P_T(t) \\
P_T(s \setminus A) &= P_T(s) \setminus A
\end{aligned}$$

$$
\begin{aligned}
P_T(s \,[\![R]\!]) &= P_T(s) \,[\![R]\!] \\
P_T(CHAOS(A)) &= CHAOS(A)
\end{aligned}
$$

Proof

Again, we show the case $\oplus = \underset{A}{\|}$. Since

$$
P_T(P \underset{A}{\|} Q) = (\mathcal{F}_T[\![P \underset{A}{\|} Q]\!], \mathcal{D}_T[\![P \underset{A}{\|} Q]\!])
$$

it suffices to show that

$$
\begin{aligned}
\mathcal{F}_T[\![P \underset{A}{\|} Q]\!] &= \pi_0(P_T(P) \underset{A}{\|} P_T(Q)) \\
\mathcal{D}_T[\![P \underset{A}{\|} Q]\!] &= \pi_1(P_T(P) \underset{A}{\|} P_T(Q))
\end{aligned}
$$

As a first step, we claim that

$$
\begin{aligned}
(s \underset{A}{\|} t)\uparrow &\Leftrightarrow s\uparrow \vee\ t\uparrow \\
(s \underset{A}{\|} t)\ ref\ X &\Leftrightarrow \exists\, Y, Z \bullet s\ ref\ Y, t\ ref\ Z, X \in Y \underset{A}{\|} Z \\
(s \underset{A}{\|} t) \stackrel{u}{\Longrightarrow} r &\Leftrightarrow \exists\, a, b \bullet s \stackrel{a}{\Longrightarrow} s', t \stackrel{b}{\Longrightarrow} t', u \in a \underset{A}{\|} b, r = s' \underset{A}{\|} t'
\end{aligned}
$$

These results follow directly from the definitions.

We consider the divergences component first

$$
\begin{aligned}
\mathcal{D}_T[\![s \underset{A}{\|} t]\!] &= \{u \mid \exists\, r \bullet s \underset{A}{\|} t \stackrel{s}{\Longrightarrow} r \wedge r\uparrow\} \\
&= \{u \mid u \in a \underset{A}{\|} b, s \stackrel{a}{\Longrightarrow} s', t \stackrel{b}{\Longrightarrow} t', (s' \underset{A}{\|} t')\uparrow\} \\
&= \{u \mid u \in a \underset{A}{\|} b, s \stackrel{a}{\Longrightarrow} s', t \stackrel{b}{\Longrightarrow} t', (s'\uparrow \vee\ t'\uparrow)\} \\
&= \left\{ \begin{array}{l} \{u \mid u \in a \underset{A}{\|} b, a \in \mathcal{D}_T[\![s]\!], t \stackrel{b}{\Longrightarrow} t'\} \\ \cup \\ \{u \mid u \in a \underset{A}{\|} b, s \stackrel{a}{\Longrightarrow} s', b \in \mathcal{D}_T[\![t]\!]\} \end{array} \right. \\
&= \pi_1(P_T(s) \underset{A}{\|} P_T(t))
\end{aligned}
$$

Similarly, for the failures component

$$
\mathcal{F}_T[\![s \underset{A}{\|} t]\!] = \left\{ \begin{array}{l} \{(u, X) \mid \exists\, r \bullet (s \underset{A}{\|} t) \stackrel{u}{\Longrightarrow} r, r\ ref\ X\} \\ \cup \\ \{(u, X) \mid \exists\, r \bullet (s \underset{A}{\|} t) \stackrel{u}{\Longrightarrow} r, r\uparrow\} \end{array} \right.
$$

$$
\begin{array}{rcl}
 & & \left\{ \begin{array}{l} \{(u,X) \mid s \stackrel{a}{\Longrightarrow} s', t \stackrel{b}{\Longrightarrow} t', u \in a \underset{A}{\parallel} b, (s' \underset{A}{\parallel} t')\ ref\ X\} \\ \cup \\ \{(u,X) \mid s \stackrel{a}{\Longrightarrow} s', t \stackrel{b}{\Longrightarrow} t', u \in a \underset{A}{\parallel} b, (s' \underset{A}{\parallel} t') \uparrow\} \end{array} \right. \\
= & & \left\{ \begin{array}{l} \{(u, Y \underset{A}{\parallel} Z) \mid s \stackrel{a}{\Longrightarrow} s', t \stackrel{b}{\Longrightarrow} t', u \in a \underset{A}{\parallel} b, s'\ ref\ Y, t'\ ref\ Z\} \\ \cup \\ \{(u,X) \mid s \stackrel{a}{\Longrightarrow} s', t \stackrel{b}{\Longrightarrow} t', u \in a \underset{A}{\parallel} b, (s' \uparrow \vee\ t' \uparrow)\} \end{array} \right. \\
= & & \left\{ \begin{array}{l} \{(u, Y \underset{A}{\parallel} Z) \mid u \in a \underset{A}{\parallel} b, (a,Y) \in \mathcal{F}_T[\![s]\!], (b,Z) \in \mathcal{F}_T[\![t]\!]\} \\ \cup \\ \mathcal{D}_T[\![s \underset{A}{\parallel} t]\!] \end{array} \right. \\
= & & \pi_0(P_T(s) \underset{A}{\parallel} P_T(t))
\end{array}
$$

□

We can now express the connection between $\mathcal{E}$ and $\mathcal{S}$, showing that the right-hand side of the diagram commutes.

Theorem 7 *For all $e \in Any$ and $\psi \in Env_T$*

$$\widehat{P_T}(\mathcal{S}[\![e]\!]\psi) \;=\; \mathcal{E}[\![e]\!](\widehat{P_T} \circ \psi)$$

Proof

Proof is by induction on e. For non-process values, this follows from the fact that the defining clauses are identical.

For process operators, we use Theorem 6 to prove the inductive step. For example

$$
\begin{array}{rcl}
\widehat{P_T}(\mathcal{S}[\![p \,\Box\, q]\!]\psi) & = & \widehat{P_T}(\mathcal{S}[\![p]\!]\psi \,\Box\, \mathcal{S}[\![q]\!]\psi) \\
 & = & \widehat{P_T}(\mathcal{S}[\![p]\!]\psi) \,\Box\, \widehat{P_T}(\mathcal{S}[\![q]\!]\psi) \\
 & = & \mathcal{E}[\![p]\!](\widehat{P_T} \circ \psi) \,\Box\, \mathcal{E}[\![q]\!](\widehat{P_T} \circ \psi) \\
 & = & \mathcal{E}[\![p \,\Box\, q]\!](\widehat{P_T} \circ \psi)
\end{array}
$$

The remaining problem is with recursive definitions

$$\widehat{P_T}(\mathcal{S}[\![\text{let } ds \text{ within } e]\!]\psi) \;=\; \widehat{P_T}(\mathcal{S}[\![e]\!]\psi')$$

where

$$\psi' \;=\; \psi \oplus \bigcup \mathcal{B}_T[\![d_i]\!]\psi'$$

so that, for some n_i and e_i

$$\begin{aligned}\widehat{P_T} \circ \psi' &= (\widehat{P_T} \circ \psi) \oplus \bigcup\{n_i \mapsto \widehat{P_T}(\mathcal{S}[\![e_i]\!]\psi')\} \\ &= (\widehat{P_T} \circ \psi) \oplus \bigcup\{n_i \mapsto \mathcal{E}[\![e_i]\!](\widehat{P_T} \circ \psi')\} \\ &= (\widehat{P_T} \circ \psi) \oplus \bigcup \mathcal{B}_{\mathcal{N}}[\![d_i]\!](\widehat{P_T} \circ \psi')\end{aligned}$$

However, this is precisely the defining equation for ρ' where

$$\mathcal{E}[\![\text{let } ds \text{ within } e]\!](\widehat{P_T} \circ \psi) = \mathcal{E}[\![e]\!]\rho'$$

From this it follows, using the result in [7] (reproduced in Appendix A) which demonstrates that the fixed point is unique, that $\rho' = \widehat{P_T} \circ \psi'$ and thus that

$$\widehat{P_T}(\mathcal{S}[\![\text{let } ds \text{ within } e]\!]\psi) = \mathcal{E}[\![\text{let } ds \text{ within } e]\!](\widehat{P_T} \circ \psi)$$

□

This is the result required to show that the right-hand side of Figure 6.1 commutes. We now proceed to combine this with the earlier results to demonstrate the congruence between the operational and denotational semantics, and thus that the full diagram commutes.

6.3 Congruence

Theorem 8 *For $e \in Any$ and $\phi \in Env_C$*

$$P_\Gamma(\mathcal{C}[\![e]\!]\phi) = \mathcal{E}[\![e]\!](\widehat{P_\Gamma} \circ \phi)$$

provided $\mathcal{E}[\![e]\!](\widehat{P_\Gamma} \circ \phi) \neq error$.

Proof

We have

$$\begin{aligned}\widehat{P_\Gamma}(\mathcal{C}[\![e]\!]\phi) &= (\widehat{P_T} \circ \widehat{\mathcal{U}})(\mathcal{C}[\![e]\!]\phi) \\ &= \widehat{P_T}(\mathcal{S}[\![e]\!](\widehat{\mathcal{U}} \circ \phi)) \\ &= \mathcal{E}[\![e]\!](\widehat{P_T} \circ \widehat{U} \circ \phi) \\ &= \mathcal{E}[\![e]\!](\widehat{P_\Gamma} \circ \phi)\end{aligned}$$

as required. The steps are justified using theorems 5, 2, 7 and 5 again. □

6.4 Summary

For the version of CSP presented, we have given both a denotational and operational semantics, layered on top of a common denotational semantics for the non-process terms. This chapter has shown that the denotational and operational semantics are congruent, connecting them via a bridging semantics defined over synchronisation trees.

That congruence depends upon only a few features of the non-process semantics. Most importantly we have the separation property

$$\mathcal{E}[\![e]\!]\rho \in E \quad \Rightarrow \quad \mathcal{E}[\![e]\!]\rho = \mathcal{E}[\![e]\!](\rho \rhd E)$$

which is required for the non-process case in Theorem 2. We also require that $error \in E$ since the semantics use this shared value to indicate type errors. Finally we require that E must have a fixed point theory; we can then use that, together with the separation property, to justify the definition of $\mathcal{B}_T$ in Section 5.2.

Moving to a richer model for the domain D_X, as mentioned in section 3.2, should be possible, provided it is possible to find an analogue of the separation property. The separation property would seem to be a sufficient condition, but perhaps not a necessary one. Such issues are interesting, but are beyond the scope of this thesis and are not discussed further here.

This concludes the theoretical results. Subsequent chapters move towards an implementation of the operational semantics.

Chapter 7

Implementing a parser

The language described so far is still a blackboard language, albeit one where the expression language is defined in more detail than is usual for CSP. This chapter takes the first steps to making the language a machine-readable notation.

The aim is to produce a grammar which is compatible with the most common parser generator, Yacc and its GNU-derivative, Bison. Although hand-crafted parsers can offer superior performance and error recovery, their construction is time-consuming and error prone.

7.1 Grammar

The grammar in Section 2.16 can be summarised as

$$
\begin{array}{rcl}
Proc & = & Common \mid \text{if } Expr \text{ then } Proc \text{ else } Proc \\
& \mid & STOP \mid SKIP \mid CHAOS(Expr) \mid Expr\ Fields \rightarrow Proc \\
& \mid & Proc \,\Box\, Proc \mid Proc \sqcap Proc \mid Proc \,;\, Proc \\
& \mid & Proc \,\triangle\, Proc \mid Proc \setminus Expr \mid Proc \,[\![Rename]\!] \\
& \mid & Proc \,|||\, Proc \mid Proc \underset{Expr}{\|} Proc \mid Proc \,{}_{Expr}\|_{Expr}\, Proc \\
& \mid & \Box\, Gens \bullet Proc \mid \sqcap\, Gens \bullet Proc \mid ;\, Gens \bullet Proc \\
& \mid & |||\, Gens \bullet Proc \mid \|_{Expr}\, Gens \bullet Proc \mid \|\, Gens \bullet (Expr, Proc) \\
& \mid & \text{let } Defns \text{ within } Proc
\end{array}
$$

Expressions are given by

$$
Expr = Common \mid \text{if } Expr \text{ then } Expr \text{ else } Expr \mid Number
$$

$$
\begin{array}{rcl}
 & | & -Expr \mid Expr + Expr \mid Expr - Expr \\
 & | & Expr * Expr \mid Expr/Expr \mid Expr\%Expr \\
 & | & \text{true} \mid \text{false} \mid Expr \wedge Expr \mid Expr \vee Expr \mid \neg\, Expr \\
 & | & Expr = Expr \mid Expr \neq Expr \mid Expr < Expr \\
 & | & Expr \leqslant Expr \mid Expr > Expr \mid Expr \geqslant Expr \\
 & | & \{Targ\} \mid \{Targ \mid Gens\} \mid \{|Targ|\} \mid \{|Targ \mid Gens|\} \mid \text{card}(Expr) \\
 & | & Expr \cup Expr \mid Expr \cap Expr \mid Expr \setminus Expr \mid \bigcup Expr \mid \bigcap Expr \\
 & | & \langle Targ \rangle \mid \langle Targ \mid Gens \rangle \mid Expr^\frown Expr \mid \text{concat}(Expr) \mid \#Expr \\
 & | & (Exprs) \mid Expr.Expr \mid \text{let } Defns \text{ within } Expr \\
 & | & \lambda\, Exprs.Expr \mid \lambda\, Exprs.Proc
\end{array}
$$

The auxiliary productions are

$$
\begin{array}{rcl}
Fields & = & \epsilon \mid Field\ Fields \\
Field & = & ?Expr \mid ?Expr : Expr \mid !Expr \\
Common & = & Name \mid Expr(Exprs) \\
Targ & = & \epsilon \mid Exprs \mid Expr..Expr \mid Expr.. \\
Gens & = & Gen \mid Gen, Gens \\
Gen & = & Expr \mid Expr \leftarrow Expr \\
Rename & = & Maps \mid Maps \mid Gens \\
Maps & = & Map \mid Map, Maps \\
Map & = & Expr \leftarrow Expr \\
Defns & = & Defn \mid Defn\ Defns \\
Defn & = & Expr = Expr \mid Proc = Proc \\
 & | & \text{channel } Names : Type \\
 & | & \text{channel } Names \\
 & | & \text{datatype } Name = VarType \\
Names & = & Name \mid Name, Names \\
Type & = & Type.Type \mid (Types) \mid Expr \mid Name \mid Name(Types) \\
VarType & = & Branch \mid Branch|VarType \\
Branch & = & Name \mid Name.Type \\
Types & = & Type \mid Type, Types
\end{array}
$$

7.2 Restructuring

This grammar cannot be used directly with parser-generators like Bison. Many productions are ambiguous. For example, *Expr* and *Proc* have productions in common which introduces ambiguity in

$$Defn \;=\; Expr = Expr \;\mid\; Proc = Proc$$

The grammar also permits nonsensical productions such as $3 \cup \langle\rangle$ which are obviously type-incorrect. It is useful to catch such errors as soon as possible, especially in the absence of a static type-checker.

This section rewrites the grammar to reduce the ambiguity and eliminate many of the type-incorrect productions. We achieve this by splitting *Expr* into elements for each of the major types, namely sets, sequences, numbers, booleans, tuples, dotted, and lambda values. We also introduce an *Any* element which is defined by

$$Any \;=\; Expr \;\mid\; Proc$$

and use it to move some of the shared productions into *Common*, this gives

$$\begin{array}{rcl}
Expr & = & Numb \mid Bool \mid Set \mid Seq \mid Tuple \mid Dotted \mid Lambda \\
Numb & = & Common \mid Number \mid \mathrm{card}(Set) \mid \#Seq \\
 & \mid & -Numb \mid Numb + Numb \mid Numb - Numb \\
 & \mid & Numb * Numb \mid Numb / Numb \mid Numb \% Numb \\
Bool & = & Common \mid \mathrm{true} \mid \mathrm{false} \\
 & \mid & Bool \wedge Bool \mid Bool \vee Bool \mid \neg\, Bool \\
 & \mid & Expr = Expr \mid Expr \neq Expr \mid Expr < Expr \\
 & \mid & Expr \leqslant Expr \mid Expr > Expr \mid Expr \geqslant Expr \\
Set & = & Common \mid \{Targ\} \mid \{Targ \mid Gens\} \\
 & \mid & \{|Targ|\} \mid \{|Targ \mid Gens|\} \\
 & \mid & Set \cup Set \mid Set \cap Set \mid Set \setminus Set \mid \bigcup Set \mid \bigcap Set \\
Seq & = & Common \mid \langle Targ \rangle \mid \langle Targ \mid Gens \rangle \\
 & \mid & Seq^\frown Seq \mid \mathrm{concat}(Seq) \\
Tuple & = & Common \mid (Exprs) \\
Dotted & = & Common \mid Expr.Expr \\
Lambda & = & Common \mid \lambda\, Exprs.Any \\
Common & = & \mathrm{if}\ Bool\ \mathrm{then}\ Any\ \mathrm{else}\ Any \\
 & \mid & Name \mid Lambda(Exprs) \\
 & \mid & \mathrm{let}\ Defns\ \mathrm{within}\ Any
\end{array}$$

The matching changes to *Proc* give

$$
\begin{array}{rcl}
Proc & = & Common \mid STOP \mid SKIP \mid CHAOS(Set) \\
& \mid & Dotted\ Fields \rightarrow Proc \\
& \mid & Proc \,\Box\, Proc \mid Proc \sqcap Proc \mid Proc \,;\, Proc \\
& \mid & Proc \,\triangle\, Proc \mid Proc \setminus Expr \mid Proc \,[\![Rename]\!] \\
& \mid & Proc \,|||\, Proc \mid Proc \underset{Set}{\|} Proc \mid Proc \,{}_{Set}\|_{Set}\, Proc \\
& \mid & \Box\, Gens \bullet Proc \mid \sqcap\, Gens \bullet Proc \mid ;\, Gens \bullet Proc \\
& \mid & |||\, Gens \bullet Proc \mid \|_{Set}\, Gens \bullet Proc \mid \|\, Gens \bullet (Set, Proc)
\end{array}
$$

Changes also need to be made to

$$
\begin{array}{rcl}
Field & = & ?Expr \mid ?Expr : Set \mid !Expr \\
Gen & = & Bool \mid Expr \leftarrow Expr \\
Targ & = & \epsilon \mid Exprs \mid Numb..Numb \mid Numb.. \\
Defn & = & Any = Any \\
& \mid & \text{channel } Names : Type \\
& \mid & \text{channel } Names \\
& \mid & \text{datatype } Name = VarType
\end{array}
$$

These changes localise the ambiguity to the multiple instances of *Common*. With some further changes, described in Appendix B, this can also be eliminated.

7.3 Representation

The basic form of input for tools is expected to take the form of a 'script' of definitions, similar to that used by functional languages like Gofer (see [21]), which is loaded into tools and then manipulated.

Simplicity and portability have a significant influence on the adoption of tools. Every additional requirement on porting the tools can limit their acceptance. The only files which are widely portable are simple ASCII text files. We therefore represent scripts using ASCII text files. More sophisticated representations are possible, but this 'lowest common denominator' representation has advantages in that it can be manipulated using existing text editors and email systems.

Expressions

Wherever possible the representation of the expression operators has been borrowed from existing languages. Set and sequence comprehensions are based on those of Haskell. Arithmetic operators are also taken from Haskell (which in turn borrowed them from C.) Comparison operators are taken directly from C. The length and catenation operators on sequence are based directly on the usual CSP representation. Boolean connectives are taken from Pascal.

The only expression operators for which no suitable existing representations could be found are those on sets. Pascal overloads the arithmetic operators ('+', '−', and '∗') for union, difference and intersection, but such overloading is confusing. For simplicity we choose to represent these using $union(a, b)$, $diff(a, b)$ and $inter(a, b)$ respectively. The distributed forms of union and intersection are represented using $Union(A)$ and $Inter(A)$.

For expressions this gives

$$
\begin{array}{rcl}
Expr & = & Numb \mid Bool \mid Set \mid Seq \mid Tuple \mid Dotted \mid Lambda \\
Numb & = & Common \mid Number \mid \underline{\texttt{card(}}Set\underline{\texttt{)}} \mid \underline{\texttt{\#}}\,Seq \\
 & \mid & \underline{\texttt{-}}\,Numb \mid Numb\,\underline{\texttt{+}}\,Numb \mid Numb\,\underline{\texttt{-}}\,Numb \\
 & \mid & Numb\,\underline{\texttt{*}}\,Numb \mid Numb\,\underline{\texttt{/}}\,Numb \mid Numb\,\underline{\texttt{\%}}\,Numb \\
Bool & = & Common \mid \underline{\texttt{true}} \mid \underline{\texttt{false}} \\
 & \mid & Bool\,\underline{\texttt{and}}\,Bool \mid Bool\,\underline{\texttt{or}}\,Bool \mid \underline{\texttt{not}}\,Bool \\
 & \mid & Expr\,\underline{\texttt{==}}\,Expr \mid Expr\,\underline{\texttt{!=}}\,Expr \mid Expr\,\underline{\texttt{<}}\,Expr \\
 & \mid & Expr\,\underline{\texttt{<=}}\,Expr \mid Expr\,\underline{\texttt{>}}\,Expr \mid Expr\,\underline{\texttt{>=}}\,Expr \\
Set & = & Common \mid \underline{\texttt{Union(}}Set\underline{\texttt{)}} \mid \underline{\texttt{Inter(}}Set\underline{\texttt{)}} \\
 & \mid & \underline{\texttt{union(}}Set\,\underline{\texttt{,}}\;Set\underline{\texttt{)}} \mid \underline{\texttt{inter(}}Set\,\underline{\texttt{,}}\;Set\underline{\texttt{)}} \mid \underline{\texttt{diff(}}Set\,\underline{\texttt{,}}\;Set\underline{\texttt{)}} \\
 & \mid & \underline{\texttt{\{}}Targ\underline{\texttt{\}}} \mid \underline{\texttt{\{}}Targ\,\underline{\texttt{|}}\,Gens\underline{\texttt{\}}} \mid \underline{\texttt{\{|}}\,Targ\,\underline{\texttt{|\}}} \mid \underline{\texttt{\{|}}\,Targ \mid Gens\underline{\texttt{|\}}} \\
Seq & = & Common \mid \underline{\texttt{<}}Targ\underline{\texttt{>}} \mid \underline{\texttt{<}}Targ\,\underline{\texttt{|}}\,Gens\underline{\texttt{>}} \\
 & \mid & Seq\,\underline{\texttt{\^{}}}\,Seq \mid \underline{\texttt{concat(}}\,Seq\,\underline{\texttt{)}} \\
Tuple & = & Common \mid \underline{\texttt{(}}\,Exprs\,\underline{\texttt{)}} \\
Dotted & = & Common \mid Expr\,\underline{\texttt{.}}\,Expr \\
Lambda & = & Common \mid \underline{\texttt{\textbackslash}}\,Exprs\,\underline{\texttt{@}}\,Any \\
Common & = & \underline{\texttt{if}}\;Bool\,\underline{\texttt{then}}\,Any\,\underline{\texttt{else}}\,Any \\
 & \mid & Name \mid Lambda\underline{\texttt{(}}Exprs\underline{\texttt{)}} \\
 & \mid & \underline{\texttt{let}}\;Defns\,\underline{\texttt{within}}\,Any
\end{array}
$$

where

$$
Common = Name \mid Expr\underline{\texttt{(}}Exprs\underline{\texttt{)}}
$$

$$
\begin{array}{rcl}
Targ & = & \epsilon \mid Exprs \mid Numb \texttt{..} Numb \mid Numb \texttt{..} \\
Gens & = & Gen \mid Gen \texttt{,} Gens \\
Gen & = & Bool \mid Expr \texttt{<-} Expr \\
Defns & = & Defn \mid Defn\ Defns \\
Defn & = & Any \texttt{=} Any \\
 & \mid & \texttt{channel}\ Names \texttt{:} Type \\
 & \mid & \texttt{channel}\ Names \\
 & \mid & \texttt{datatype}\ Name \texttt{=} VarType \\
Names & = & Name \mid Name \texttt{,} Names \\
Type & = & Type \texttt{.} Type \mid \texttt{(} Types \texttt{)} \mid Expr \mid Name \mid Name \texttt{(} Types \texttt{)} \\
VarType & = & Branch \mid Branch \texttt{|} VarType \\
Branch & = & Name \mid Name \texttt{.} Type \\
Types & = & Type \mid Type \texttt{,} Types
\end{array}
$$

Processes

The representation of the process syntax was chosen to resemble the typeset form of the CSP operators, within the restrictions imposed by the ASCII character set.

$$
\begin{array}{rcl}
Proc & = & Common \mid \texttt{STOP} \mid \texttt{SKIP} \mid \texttt{CHAOS(}Set\texttt{)} \\
 & \mid & Dotted\ Fields \texttt{->} Proc \\
 & \mid & Proc \texttt{[]} Proc \mid Proc \texttt{|~|} Proc \mid Proc \texttt{;} Proc \\
 & \mid & Proc \texttt{/\textbackslash} Proc \mid Proc \texttt{\textbackslash} Expr \mid Proc \texttt{[[} Rename \texttt{]]} \\
 & \mid & Proc \texttt{|||} Proc \mid Proc \texttt{[|} Set \texttt{|]} Proc \mid Proc \texttt{[} Set \texttt{||} Set \texttt{]} Proc \\
 & \mid & \texttt{[]}\ Gens \texttt{@} Proc \mid \texttt{|~|}\ Gens \texttt{@} Proc \mid \texttt{;}\ Gens \texttt{@} Proc \\
 & \mid & \texttt{|||}\ Gens \texttt{@} Proc \mid \texttt{[|}\ Set \texttt{|]}\ Gens \texttt{@} Proc \mid \texttt{||}\ Gens \texttt{@} \texttt{[}Set\texttt{]}\ Proc \\
Fields & = & \epsilon \mid Field\ Fields \\
Field & = & \texttt{?}Expr \mid \texttt{?}Expr \texttt{:} Expr \mid \texttt{!}Expr \\
Rename & = & Maps \mid Maps \texttt{|} Gens \\
Maps & = & Map \mid Map \texttt{,} Maps \\
Map & = & Dotted \texttt{<-} Dotted
\end{array}
$$

7.4 Precedence

The next issue that must be resolved before a grammar suitable for use with Bison can be constructed is the precedence of the operators.

Expressions

Given that the representation for most of the expression operators was borrowed from other languages, it is natural to borrow the relative precedence from those languages.

The precedence and associativity of the arithmetic and comparison operators is as given in [17] and is shown in table 7.1.

Operator	Description	Associativity
$-$	unary minus	
$*$, $/$, $\%$	multiplicative	left
$+$, $-$	additive	left
$<$, $>$, $<=$, $>=$	comparison	left
$==$, $!=$	equality	left

Table 7.1: Basic precedences in C

Following C again, but unlike Pascal, the logical operations are given a lower precedence than the comparisons. Negation binds most tightly, then conjunction binds more tightly than disjunction.

Because of the structure of the grammar, the sequence operations are largely separate; it is simply necessary that they have a higher precedence than the logical operations. We make the length operation bind less tightly than catenation so that $\#s^\frown t$ means the same as $\#(s^\frown t)$.

Since it is less common to compare events, as in $(c.1) == (c.2)$ than it is to allow logical values as part of protocols, as in $c.(1 == 2)$ we make the dot operation bind less tightly than the logical operations.

Conditionals, lambda and let expressions are all given a low priority so that the body of the expression extends as far as possible. At the opposite extreme, function application binds most tightly to match the precedence of this in most other languages.

Processes

A quick survey of the CSP community within Oxford demonstrated early in this project that, while there was general agreement on some basic precedences, there was no definitive opinion when it came to the obscure cases. Indeed, it sometimes seemed that the precedence used in published papers was the 'obvious' one implied by the context of the process and the intention of the author.

An inspection of [18] makes it clear that the language presented there needs very few precedence rules at all; cases which might otherwise be ambiguous are carefully supplied with bracketing. Many of the theoretical papers on CSP also have no need to discuss operator precedence; the grammar is taken as given, the functions are being defined on the trees produced from the grammar, and process expressions involve only a single operator since functions are defined on the syntax one operator at a time.

It is clear that the precedence of the operators should reflect the typical structure of processes. In occam for example, the choice construct is necessarily built from prefixed processes and it is common to decompose a system as a network of long-lived parallel components, reserving sequential behaviour and choices for the implementation of the components themselves.

Similarly, work by D. Jackson (see [20]) describes the basic properties of a system using small sequential processes which can then be combined using the choice and parallel operations. Similar work on the specification of the T9000 VCP by Geoff Barrett and others at Inmos, described in [3] and [4], structures specifications in precisely the same way; this in turn led to the two-level approach used in FDR 1.4 where processes are built using sequential and choice operations, with parallelism being restricted to an outer level.

Such a parallel network of choices between sequential components is typical of the structure of CSP processes. This gives the basic operator precedence shown in table 7.2.

Operator	Description	Associativity
$\rightarrow$	prefix	
;	sequence	left
$\Box$	external choice	left
$\sqcap$	internal choice	left
$\|\|\|$	interleave	left
$\|\|_A$, $\|\|_B$	parallel	none

Table 7.2: Process operator precedence

Since the interrupt operation has a behaviour between those of sequential composition and external choice, it is reasonable to give it a precedence between those operators.

The hiding operation usually occurs at the outermost level, concealing events used to connect processes in parallel. It thus makes sense to give hiding a precedence lower than any of the other operations, but this is rarely exploited in examples; it seems much more common to use bracketing to make the extent of the hiding clear, or to explicitly name the network before applying the hiding operation.

The replicated operations are given a still lower precedence so that the body of the expression extends as far right as possible. Conversely, the body of the renaming operation should be as small as possible so this is given a high precedence.

7.5 Summary

This chapter has taken the grammar of Chapter 2 and produced a related grammar which has an ASCII representation for the symbols and is close to being implementable using standard parser-generators. The remainder of that implementation is given in Appendix B. The next chapter will assume the existence of a parser and discuss the implementation of the operational semantics.

Chapter 8

Implementing the semantics

We now outline an implementation of the operational semantics. This allows exploration of processes as part of tools such as refinement checkers and simple animation tools. Moving to such a level of detail is, of course, probably too operational for tools such as proof assistants or translation utilities.

As part of this process, it is necessary to implement the expression language supporting the process language, which is essentially a lazy functional programming language in its own right.

The earliest attempt at providing such an implementation was that underlying the compilation engine of FDR 1.4. It was implemented as a translation from a simple syntax tree into Standard ML, the ML then being merged into the running FDR engine. This provides a useful benchmark against which other implementations may be measured.

8.1 Implementation language

The choice of programming language in which to implement the operational semantics is influenced by the portability, readability, efficiency, and extensibility of the resulting system.

The only suitable imperative programming languages which are available on a wide range of platforms are ANSI-C and C++. Despite C's superior portability, it was decided that C++ was a more suitable language for the implementation. Its stronger type system and support for object-oriented programming outweighed the fact that the language was in a state of flux, pending the ISO standardisation.

An implementation in Java would have been significantly simpler, but Java was

not available when work on the implementation began. Java is also less efficient and less widely available than C++.

Throughout what follows, the reader is assumed to have some familiarity with with the features of C++ described in [37]. Familiarity with a standard imperative language may suffice, given access to such a text.

8.2 Representing a branched type

The types needed to encode the syntax tree are implied by the grammar, and can be naturally expressed as a Gofer style type definition. The best way of encoding such a branched type in C++ defines each branch as a separate class. Virtual methods, declared in a common base-class and implemented by the subclasses provide the operations on the type. Thus,

```
class T_base
{
public:
  virtual int f() = 0; // No default implementation
};

class T_A : public T_base
{
  int x, y;
public:
  T_A(int m, int n) { x = m; y = n; }
  int f() { return x + y; }
};

class T_B : public T_base
{
  int x;
public:
  T_B(int n) { x = n; }
  int f() { return 2 * x; }
};

int f(T_base *t)
{ return t->f(); }
```

can be regarded as an implementation of

```
T = A Int Int | B Int
f :: T -> Int
f(A x y) = x + y
f(B x)   = 2 * x
```

Reference counting

An unfortunate feature of this approach is that pointers (or references) must be used, rather than items themselves; this is mainly an artifact of the way C++ implements dynamic binding. We use a pair of classes, `Pointer` and `Counter`, to encapsulate the pointers. The abstract base-classes are all derived from `Counter`, while `Pointer` is used to provide a helper class which corresponds to the basic type and forwards all the defined operations. This has the advantage of giving us a mechanism for managing the free-store.

Given these basic pieces of machinery, the earlier example becomes

```
class T_base : public Counter
{
public:
  virtual int f() = 0;
};

// As before

class T_A : public T_base
{
  int x, y;
public:
  T_A(int m, int n) { x = m; y = n; }
  int f() { return x + y; }
};

class T_B : public T_base
{
  int x;
public:
  T_B(int n) { x = n; }
  int f() { return 2 * x; }
};
```

```
// Hide the pointers

class T
{
  Pointer<T_base> ptr;
  T(T_base *x)
    : ptr(x)
  { }
public:
  friend T A(int m, int n)
  { return T(new T_A(m, n)); }
  friend T B(int n)
  { return T(new T_B(n)); }

  int f()
  { return ptr->f(); }
};
```

where we use the functions `A` and `B` to build the branches.

8.3 The syntax tree

Encoding the syntax tree in the style presented above gives

```
class Syntax
{
  Pointer<const sBase> ptr;
public:
  friend ostream &operator<<(ostream &, const Syntax &);

  // Operations on syntax to be defined later
};
```

where the base class for the syntax hierarchy is defined as

```
class sBase : public Counter
{
public:
  virtual ostream &output(ostream &) const = 0;
```

```
  // Other operations described later
};
```

The majority of the member functions for this class have been omitted and will be introduced later. However, the mechanism used to allow output of syntactic values to a stream is shown; the output operation on `Syntax` class is defined in terms of the virtual output method of the base class.

```
ostream &operator<<(ostream &os, const Syntax &x)
{ return x.ptr->output(os); }
```

This illustrates the general technique used to implement operations on syntactic values, explicitly passing the call through a member function of the `Syntax` class to a virtual method is the class hierarchy. It is worth noting that most of the work involved in developing smart-pointers and forwarding calls would not have been necessary in a language such as Java, where such details are effectively part of the object model.

These fragments also indicate some of the conventions adopted throughout what follows. The output operator will always be defined using a virtual method of the hierarchy. Classes which belong to a specific hierarchy will have names beginning with a lower case letter indicating which hierarchy they belong to; this is especially important in the syntax and value hierarchies, since we need to model and clearly separate corresponding branches in these spaces. Finally, wherever possible, code will be in a functional style, so that methods return results rather than acting by side-effect. Hence in the above fragment `Pointer<const sBase>` can be used since none of the methods of `sBase` have side-effects.

Since the entire interface of any syntax node is defined through `sBase`, we can avoid placing the class definitions in the header files; all we need is a method of constructing the classes. For example in a header file we might have

```
Syntax sName(const char *);
```

which declares the function which returns a name node for the syntax tree and we implement this using

```
class sName_ : public sBase
{
  Text text;
public:
  sName_(const char *s)
```

```
    : text(s)
  { }
};

Syntax sName(const char *s)
{
  return new sName_(s);
}
```

where the class name uses a trailing underscore to indicate that the name is private. This separation of interface and implementation keeps the header files small and reduces the compilation time due to dependencies.

In precisely the same way we can encode the other types needed to represent the full tree, including `Field`, `Target` and `Generator`. Operations on these types, as well as on `Syntax`, will be introduced as needed to support the implementation of the semantics in the following sections. These functions should also make it clear why it is helpful to separate these three special cases from the large `Syntax` space.

Providing operations as virtual methods on classes representing the syntax makes this implementation an instance of the Interpreter pattern described in [15].

8.4 Representing values

Although the denotational semantics makes use of both expression and process values, with expression values further broken down into numbers, sets, sequences, and so on, the semantic function yields values which lie in $D_{\mathcal{N}}$, and in the same way we require a general notion of values which can be any of these.

We use the same technique we used to model the branched type for the syntax tree, but because of the way the space breaks down we will make much more use of intermediate classes. We will call the root of the hierarchy `vBase`, and `Value` will effectively be a pointer to a subclass of `vBase`. (The mechanics involved in efficiently implementing shared-reduction require the introduction of another level of indirection, but the details are omitted.)

We avoid an implementation in the style of [27]. Although the super-combinator approach has performance advantages, the more direct encoding used here is more suitable when language features may need to be quickly prototyped. The performance overhead is roughly one order of magnitude over the excellent Gofer implementation, but it is hoped that the code here is more suitable for experimentation. In particular, it is far from clear how to add process values into a Gofer style im-

plementation and it is unlikely that in any tool reasoning about CSP processes the evaluation of the expression-language will be a major bottle-neck.

Reduction

The first split in the value hierarchy is between values which are in normal form and those which are not. Values are in normal form if they have been reduced to the point at which it is possible to identify to which of the sub-domains of $D_{\mathcal{N}}$ they belong. For example we class a cons-cell (used to build sequences) as being in normal form, even if the tail of the sequence has not been reduced. This corresponds to the notion of weak-head normal forms in more usual implementations (again, see [27].)

Values are reduced to normal form by the method `reduce` which returns the normal form of a given value, so the base class has

```
class vBase : public Counter
{
public:
  // Reduce
  virtual Value reduce() const = 0;
  // Many more functions omitted
};
```

We then split this into

```
class vNormal : public vBase
{
public:
  Value reduce() const { return Value(this); }
};
```

since values which are already in normal form can be returned directly, and

```
class vNotNormal : public vBase
{
public:
  // reduce defined by subclasses, so remains abstract
};
```

so that reduction of a value will either return the value itself or force the reduction to occur. (In fact the base class for values has many more methods, most of which

make sense only for values in normal form and the implementations of these in `vNotNormal` force the reduction when these methods are called.)

An example of a value which is not in normal form is given by a closure, which is simply a piece of syntax together with an environment in which that syntax is to be evaluated. The first step in the reduction of such a closure must obviously be controlled by the structure of the syntax from which it is built, so we require a method in the syntax hierarchy to allow us to perform this step.

```
class sBase : public Counter
{
public:
  // One function out of many ...
  virtual Value value(const Env &) const = 0;
};
```

Given this support, closures are written trivially as

```
class vClosure_ : public vNotNormal
{
  Syntax s;
  Env e;
public:
  Value reduce() const
  { return s.value(e).reduce(); }
};
```

where the result obtained from `value` is not guaranteed to be in normal form, simply to be 'closer' to normal form.

The implementations of the value method in the syntax hierarchy correspond exactly to the clauses in the denotational semantics which translate syntactic operations into semantic ones. For example, names are looked up in the environment

```
class sName_ : public sBase
{
  Text text;
public:
  Value value(const Env &e) const
  { return e.lookup(text); }
};
```

Similarly, unary minus forces evaluation of the argument, converts the general value that result to a value which is guaranteed to be numeric, negates that and returns the result.

```
class sUMinus_ : public sBase
{
  Syntax arg;
public:
  Value value(const Env &e) const
  { return vNumber(-arg.value(e).number()); }
};
```

If the value is not numeric, then the call to `number` does not return. Ideally, the function would throw an exception indicating that a type error had occurred.

8.5 Environments

An environment associates names with values. This is implemented as a linked list of pairs. We can add elements point-wise using

```
Env Env::add(const Text &, const Value &) const;
```

and we can also add an existing environment

```
Env Env::add(const Env &) const;
```

and we can then retrieve elements using

```
Value Env::lookup(const Text &) const;
```

Adding a list of definitions is more complex and will be considered in the following sections.

Definitions

While the above methods for extending a binding are useful, the main way names are bound is using (mutually recursive) definitions. A basic definition of the form

$$x \quad = \quad y + 2$$

associates the syntactic term $y + 2$ with the identifier x. To obtain a *value* to associate with x, the syntactic term must be paired with an environment (to form a closure), and since the definition may be recursive, this in general needs to be the environment which is under construction.

This suggests that definitions should be able to provide an association between names and syntactic terms (which we will call a `PreEnv`) so we have

```
class dBase : public Counter
{
public:
  virtual PreEnv change() const;
};
```

where a pre-environment is similar to an environment

```
class PreEnv
{
  typedef Pair<Text, Syntax> pair;
  typedef List<pair> plist;
  plist p;
public:
  PreEnv add(const Text &, const Syntax &) const;
  PreEnv add(const PreEnv &) const;

  // Traverse the list of pairs
  int null() const;
  pair car() const;
  PreEnv cdr() const;
};
```

Binding syntax

Basic definitions provide a pre-environment based on the structure of the syntax on the left-hand side, so we have

```
class dTerm_ : public dBase
{
  Syntax lhs, rhs;
public:
  PreEnv change() const
```

```
  { return lhs.bind(rhs); }
};
```

with much of the work transferred to the `bind` method of the syntax classes. Only a small proportion of syntactic terms may be used on the left-hand side of a basic definition, so the default implementation of the binding method indicates that the binding operation is not supported.

```
PreEnv sBase::bind(const Syntax &t) const
{ illegal("Bad binding attempt"); }
```

The two required implementations are for simple names

```
PreEnv sName_::bind(const Syntax &t) const
{ return PreEnv().add(text, t); }
```

and functional applications which are equivalent to definitions involving lambda terms (since $f(x) = y$ and $f = \lambda\, x.y$ are equivalent)

```
PreEnv sApply_::bind(const Syntax &t) const
{ return lhs.bind(sLambda(rhs, t)); }
```

Unrolling

Given all this support, adding definitions to an environment can be written as

```
Env Env::add(const DefnList &x) const
{
  Env result    = *this;
  PreEnv normal = unroll(x);
  result.add(normal);
  result.patch(normal);
  return result;
}
```

where the first step builds the pre-environment from the list of definitions, the second step adds this to an environment using dummy empty values, and finally the empty values are patched with closures using the resulting environment.

The 'knot' tied in the environment when adding a set of definitions is sufficient to prevent the reference counting scheme from working correctly, but the leak this causes has proved to be minor.

The actual implementation also checks that identifiers are not bound repeatedly, but those details are omitted here. These checks reject scripts such as

$$\begin{aligned} x &= 2 \\ x &= 3 \end{aligned}$$

Local definitions

Given this method for extending environments, the class which supports local definitions ('let' clauses) can be defined trivially as

```
class sLet_ : public sBase
{
  DefnList ds;
  Syntax e;
public:
  Value value(const Env &env) const
  { return e.value(env.add(ds)); }
};
```

8.6 Applications

Function application is recorded in the syntax using

```
class sApply_ : public sBase
{
  Syntax lhs;
  SyntaxList rhs;
public:
  Value value(const Env &e) const
  { return vApply(vClosure(lhs, e), vClosure(sTuple(rhs), e)); }
};
```

where the first reduction step converts the construct into an application of one value to another. To reduce the resulting value, the left-hand side must be a lambda term which can be applied to the right-hand side. Hence

```
class vApply_ : public vNotNormal
{
```

```
  Value lhs, rhs;
public:
  Value reduce() const
  { return lhs.lambda().apply(rhs).reduce(); }
};
```

If the only applications resulted from those in the syntax, we could implement the `value` operation of `sApply_` directly, without requiring `vApply_`. However, the application of one value to another occurs in other places in the semantics.

Lambda terms

A lambda term is represented in the syntax tree using

```
class sLambda_ : public sBase
{
  SyntaxList args;
  Syntax body;
public:
  Value value(const Env &e) const
  { return vLambda(sTuple(args), body, e); }
};
```

The `value` operation simply converts the syntactic lambda term into a value by adding in the environment in which the term was built. This, along with the corresponding clause for applications, corresponds to the decision to use static scoping.

The resulting value simply stores all three components until application

```
class vLambda_ : public vNormal
{
  Syntax args;
  Syntax body;
  Env env;
public:
  VLambda lambda() const;
  // Specific methods
  Value apply(const Value &) const;
};
```

The result of the application is simply the body of the lambda term in the environment formed from that supplied by binding the arguments of the term to the value given. Hence

```
Value vLambda_::apply(const Value &v) const
{ return vClosure(body, env.add(args.bind(v))); }
```

Binding values

The clause above requires a mechanism for binding syntax to values giving an environment as a result. (The method of Section 8.5 binds syntax to syntax giving a pre-environment.) This is simply another virtual method in the syntax hierarchy.

```
class sBase : public Counter
{
public:
  virtual Env bind(const Value &) const;
};
```

which is implemented for names and tuples. The default implementation is to report an error

```
Env sBase::bind(const Value &v) const
{ illegal("Bad binding attempt"); }
```

Again, the actual implementation checks that the same identifier is not bound repeatedly.

8.7 Booleans

The basic property of logical values is that they can be used to choose between two alternatives, thus the fundamental operations are

```
Value VLogic::cond(const Value &, const Value &y) const;
Value VLogic::cond(const Syntax &, const Syntax &) const;
```

Given the switching mechanism of logical values, conditional expressions (if/then/else clauses) can be encoded simply as

```
class sCond_ : public sBase
{
  Syntax g, t, f;
public:
  Value value(const Env &e) const
  { return g.value(e).logic().cond(t, f).value(e); }
};
```

Comparisons

The comparison operations are defined on numbers, sets and sequences and are thus overloaded. The precise function called depends upon the exact type of the arguments. The natural way of implementing this in C++ is to use virtual methods. It is, however, excessive to provide a virtual method for each comparison operation, since they are strongly related. The `Value` hierarchy therefore provides two basic tests, the first for equality and the second for less-than-or-equality.

```
class vBase : public Counter
{
public:
  virtual int eq(const Value &) const;
  virtual int le(const Value &) const;
};
```

with default implementations that indicate that the operation is not supported.

In the case of logical values themselves, equality is defined but no ordering is supplied.

```
int vLogic_::eq(const Value &x) const
{ return x == b.logic(); }
```

8.8 Numbers

Values

In the same way that `VLogic` captures the properties we require of boolean values, `VNumber` captures the properties of numeric values including arithmetic operations and comparisons. The behaviour of the numeric type was outlined in Section 3.5 in such a way that it is possible to use a 32bit C++ integer as the underlying

implementation, but since the precise representation is hidden behind `VNumber`, it is easy to use an extended integer type if required.

```
class VNumber
{
  int n;
public:
  friend VNumber operator+(const VNumber &x, const VNumber &y);
  // Other arithmetic operations

  int operator==(const VNumber &x) const
  { return n == x.n; }
  // Other comparisons
};
```

A simple wrapper embeds this in the `Value` hierarchy

```
class vNumber_ : public vNormal
{
  VNumber n;
public:
  int eq(const Value &) const;
  int le(const Value &) const;
};
```

where the virtual methods are trivially defined.

8.9 Sequences

Although the syntax used for sequences is based on that of CSP, the details are closer to those of the lists found in Orwell and Gofer. However, the underlying implementation is based on Lisp's classic cons-cells where

```
List a = Cons a (List a) | Nil
```

with fundamental operations

```
car  : (List a)->a
cdr  : (List a)->(List a)
null : (List a)->Bool
```

This is yet another branched type, so making these three operations virtual methods across the branches (and using the Gofer names rather than those from Lisp) suggests

```
class vSeqBase : public vNormal
{
public:
  virtual int null() const = 0;
  virtual Value head() const = 0;
  virtual Value tail() const = 0;
};
```

which has precisely two subclasses

```
class vNil_ : public vSeqBase
{
public:
  vNil_();
  // Obvious implementations of null/head/tail
};
```

and

```
class vCons_ : public vSeqBase
{
  Value a, b;
public:
  vCons_(const Value &, const Value &);
  // Obvious implementations of null/head/tail
};
```

Both of these classes are hidden behind `VSeq` which is essentially a pointer to `vSeqBase_`, forwarding on the head, tail, and null operations.

The tail operation returns a general value rather than a sequence since otherwise we are unable to support full lazy evaluation. The head, tail and null operations are connected to the operations of the same name in the expression language by simple veneers.

Since sequences are also ordinary values, it is necessary to provide comparisons upon them. These can be implemented at the level of `vSeqBase` using the head, tail and null operations.

Length and membership tests are not lazy and so can be coded directly. The member operations `length` and `contains` can be implemented simply in terms of head, tail and null in `vSeqBase`.

Catenation of sequences is slightly more complex, since the operation must be lazy. This easiest way of supporting this is with a simple class.

```
class vCat_ : public vNotNormal
{
  Value lhs, rhs;
public:
  Value reduce() const
  {
    VSeq x = lhs.seq();
    return x.null()
         ? rhs.reduce()
         : vCons(x.head(), vCat(x.tail(), rhs));
  }
};
```

Concatenation can then be expressed using a similar class which expands into simple catenations.

8.10 Sets

As suggested by the set model used Chapter 3, the set implementation operates at two basic levels. The simplest is a basic concrete representation with a definite enumeration order. The approach used in the semantics was a potentially infinite sequence of non-empty finite sets. That is simplified here to a potentially infinite, unordered sequence of values, without duplicates. This yields an implementation similar to that of sequences, the two cases being a 'cons-cell' and the 'nil-list'. In the case of concrete sets, these are called `vGlue` and `vEmpty`, these being instances of the class

```
class vCSetBase : public vSetBase
{
public:
  // Additional functionality for concrete sets
  virtual Value pick() const = 0;
  virtual Value rest() const = 0;
};
```

It is undesirable to encode sets as ordered lists since there is no natural order on some of the elements from which sets might be built.

The structured sets used in the semantics are derived directly from `vSetBase` and so lack the enumeration interface provided by the `pick` and `rest` operations. The more abstract form of sets which this class represents allows tests to see if the set is empty, counting of the elements, and a membership test.

```
class vSetBase : public vNormal
{
public:
  virtual int empty() const = 0;
  virtual VNumber cardinality() const = 0;
  virtual int has_member(const Value &) const = 0;
};
```

8.11 Comprehensions

While Section 8.9 and Section 8.10 have discussed the representation of sequences and sets, along with the operations defined upon them, the comprehensions which are the basic way of constructing them have still to be covered. Comprehensions are built from two components, the target indicating the results, and a list of generators controlling the expansion.

The base case for the comprehensions occurs when the generator list is empty, as in $\langle 1, 2, 3 \rangle$ and $\{7..\}$. The element of the syntax tree corresponding to these are

```
SeqComp(tEnum([1,2,3]), GenerateList())
```

and

```
SetComp(tUpFrom(7), GenerateList())
```

respectively. Since the comprehension term does not have access to the precise kind of target which is involved, the code which converts the target into the appropriate type must be associated with the target itself. This implies that targets must be capable of producing either a sequence or a set on demand.

Targets

Targets come in three basic flavours: simple enumerations, closed ranges, and open ranges. Again, we model the branches by derivation from a base class with virtual

methods.

```
class tBase : public Counter
{
public:
  virtual Value seq(const Env &) const = 0;
  virtual Value set(const Env &) const = 0;
};
```

An enumerated target simply takes the syntax list from which it was built, forms a list of closures and converts that to the required construct.

Open ranges are infinite structures and so are converted to classes which hold the delayed part of the evaluation.

```
class vUpFrom_ : public vNotNormal
{
  VNumber base;
  AddOp add;
public:
  Value reduce() const
  { return add(vNumber(base), new vUpFrom_(base+1, add)); }
};
```

with this used by

```
Value vSetUpFrom(const VNumber &n)
{ return new vUpFrom_(n, vGlue); }
```

and

```
Value vSeqUpFrom(const VNumber &n)
{ return new vUpFrom_(n, vCons); }
```

Closed ranges expand into a similar class, with the addition of an upper bound and another function argument to build the empty version of the result. In fact, it is possible to optimise both of the set range classes by using an approach similar to that of Chapter 3, making the ranges special set constructors and optimising the membership tests and cardinality calculations, but that is not discussed further here.

Generators

As in the semantic clauses, generators can be rewritten using

$$\begin{array}{rcl} \{t \mid b, gs\} & \equiv & \text{if } b \text{ then } \{t \mid gs\} \text{ else } \{\} \\ \{t \mid e_1 \leftarrow e_2, gs\} & \equiv & \bigcup\{\{t \mid gs\} \mid e_1 \leftarrow e_2\} \end{array}$$

with similar equivalences holding for sequences.

In both cases, it suffices to call a method of the generator indicating whether a set or sequence result is required along with the simpler comprehension $\{t \mid gs\}$. Hence we take

```
class gBase : public Counter
{
public:
  virtual Syntax seq(const Syntax &) const = 0;
  virtual Syntax set(const Syntax &) const = 0;
};
```

Filter operations are then encoded as

```
class gFilter_ : public gBase
{
  Syntax g;
public:
  Syntax set(const Syntax &x) const
  {
    return sCond(g, x,
      sSetComp(tEnum(SyntaxList()), GenerateList()));
  }
  // Similarly for sequences
};
```

where `sCond` is the syntax-tree encoding for if/then/else expressions.

The productive generators are slightly more complex, since they make use of equivalences

$$\{e \mid e_1 \leftarrow e_2\} \equiv \mathit{mapset}(\lambda\, e_1.e, e_2)$$

to eliminate the comprehension.

Unrolling

Given this support from the targets and generators, set comprehensions are implemented using

```
class sSetComp_ : public sBase
{
  Target t;
  GenerateList g;
public:
  Value value(const Env &e) const
  {
    return g.null()
         ? t.set(e)
         : g.car().set(sSetComp(t, g.cdr())).value(e);
  }
};
```

with a similar form used for sequence comprehensions.

8.12 Processes

The operational semantics of Chapter 4 characterised the process component using a transition relation $\rightarrow$ defined over Σ_τ between configurations in Γ, where

$$\begin{array}{rcl} \Gamma & \supset & C \cup \{error\} \\ \rightarrow & \subseteq & \Gamma \times \Sigma_\tau \times \Gamma \end{array}$$

Since every element of C is a pair consisting of process syntax and an environment, C can be represented (as closures) within the existing space of values. Given a specific process P, the relevant component of the transition relation is

$$(P \rightarrow) \quad \subseteq \quad \Sigma_\tau \times \Gamma$$

While it is possible to calculate this relation directly (as a list of pairs, say), the resulting value can be large. Rewriting the relation to the equivalent form

$$after(P) \quad = \quad \Sigma_\tau \rightarrow \mathbb{P}(\Gamma)$$

allows us to expand the process pointwise, using much less space. However, it is often useful to know which events a process can participate in before considering

the behaviour after those events. Calculating this from a function like that above involves an expression like $\{e \in \Sigma_\tau \mid after(P, e) \neq \{\}\}$ which involves exploring the (potentially large) space Σ_τ for all processes. This can be avoided by supporting both operations directly

$$\begin{aligned} inits(P) &= \{e \in \Sigma_\tau \mid \exists\, Q \bullet P \xrightarrow{e} Q\} \\ after(P, e) &= \{Q \mid P \xrightarrow{e} Q\} \end{aligned}$$

so that

$$e \in inits(P) \quad \Leftrightarrow \quad after(P, e) \neq \{\}$$

Process values thus have two specific methods, corresponding to these functions. However, it is not possible for `after` to return a set of processes. A definition of process equality (which cannot be supplied) would be necessary to construct the lists without duplicates which are the underlying representation for sets. It is possible to add another form of set (with duplicates) to the value hierarchy, but the resulting class is indistinguishable from the sequences it already provides. Hence, it is simpler to use sequences as the return value from the `after` function.

Given this, it is simple to translate from then *inits* and *after* functions to code, and we will simply give the definitions for those functions, omitting the code.

Basic processes

The simplest process definition is that for $STOP$ which has no initial actions

$$\begin{aligned} inits(STOP) &= \{\} \\ after(STOP, e) &= \{\} \end{aligned}$$

The process $SKIP$ is slightly more complicated; the only initial action is $\checkmark$, after which it behaves like $STOP$

$$\begin{aligned} inits(SKIP) &= \{\checkmark\} \\ after(SKIP, e) &= \begin{cases} \{STOP\} & \text{, if } e = \checkmark \\ \{\} & \text{, otherwise} \end{cases} \end{aligned}$$

Sequence

For sequential composition,

$$inits(P\ ;\ Q) = \begin{cases} inits(P) & \text{, if } \checkmark \notin \text{inits}(P) \\ (inits(P) \setminus \{\checkmark\}) \cup \{\tau\} & \text{, otherwise} \end{cases}$$

$$after(P\,;Q,e) \;=\; \left\{\begin{array}{l} \{P'\,;Q \mid e \neq \checkmark \wedge P' \in after(P,e)\} \\ \cup \\ \{Q \mid e = \tau \wedge \checkmark \in inits(P)\} \end{array}\right.$$

Interrupt

The interrupt operator behaves like a mixture of external choice and sequential composition; the left component is allowed to perform arbitrary actions without eliminating the operator, but any visible action by the right component reduces the expression to that component.

$$\begin{array}{rcl} inits(P \mathbin{\triangle} Q) & = & inits(P) \cup inits(Q) \\ after(P \mathbin{\triangle} Q, e) & = & \left\{\begin{array}{l} \{P' \mathbin{\triangle} Q \mid P' \in after(P,e)\} \\ \cup \\ \{P \mathbin{\triangle} Q' \mid Q' \in after(Q,e), e = \tau\} \\ \cup \\ \{Q' \mid Q' \in after(Q,e), e \neq \tau\} \end{array}\right. \end{array}$$

Choice

For the choice operators we have

$$\begin{array}{rcl} inits(P \sqcap Q) & = & \{\tau\} \\ after(P \sqcap Q, e) & = & \{P, Q \mid e = \tau\} \\ inits(P \mathbin{\Box} Q) & = & inits(P) \cup inits(Q) \\ after(P \mathbin{\Box} Q, e) & = & \left\{\begin{array}{ll} after(P,e) \cup after(Q,e) & \text{, if } e \neq \tau \\ \left(\begin{array}{c} \{P' \mathbin{\Box} Q \mid P' \in after(P,e)\} \\ \cup \\ \{P \mathbin{\Box} Q' \mid Q' \in after(Q,e)\} \end{array}\right) & \text{, otherwise} \end{array}\right. \end{array}$$

Parallel

The initial actions for the parallel operators are given by

$$inits(P \mid\mid\mid Q) \;=\; \left\{\begin{array}{l} inits(P) \setminus \{\checkmark\} \\ \cup \\ inits(Q) \setminus \{\checkmark\} \\ \cup \\ inits(P) \cap inits(Q) \cap \{\checkmark\} \end{array}\right.$$

$$
\begin{aligned}
inits(P \underset{A}{\parallel} Q) &= \left\{ \begin{array}{l} inits(P) \setminus A_\surd \\ \cup \\ inits(Q) \setminus A_\surd \\ \cup \\ inits(P) \cap inits(Q) \cap A_\surd \end{array} \right. \\
inits(P \ {}_A\|_B \ Q) &= \left\{ \begin{array}{l} inits(P) \cap (A_\tau \setminus B) \\ \cup \\ inits(Q) \cap (B_\tau \setminus A) \\ \cup \\ inits(P) \cap inits(Q) \cap (A \cap B)_\surd \end{array} \right.
\end{aligned}
$$

and similarly

$$
\begin{aligned}
after(P \mid\mid\mid Q, e) &= \left\{ \begin{array}{l} \{P' \mid\mid\mid Q \mid P' \in after(P, e), e \neq \surd\} \\ \cup \\ \{P \mid\mid\mid Q' \mid Q' \in after(Q, e), e \neq \surd\} \\ \cup \\ \{P' \mid\mid\mid Q' \mid P' \in after(P, e), \\ \qquad Q' \in after(Q, e), e = \surd\} \end{array} \right. \\
after(P \underset{A}{\parallel} Q, e) &= \left\{ \begin{array}{l} \{P' \underset{A}{\parallel} Q \mid P' \in after(P, e), e \notin A_\surd\} \\ \cup \\ \{P \underset{A}{\parallel} Q' \mid Q' \in after(Q, e), e \notin A_\surd\} \\ \cup \\ \{P' \underset{A}{\parallel} Q' \mid P' \in after(P, e), \\ \qquad Q' \in after(Q, e), e \in A_\surd\} \end{array} \right. \\
after(P \ {}_A\|_B \ Q, e) &= \left\{ \begin{array}{l} \{P' \ {}_A\|_B \ Q \mid P' \in after(P, e), e \in A_\tau \setminus B\} \\ \cup \\ \{P \ {}_A\|_B \ Q' \mid Q' \in after(Q, e), e \in B_\tau \setminus B\} \\ \cup \\ \{P' \ {}_A\|_B \ Q' \mid P' \in after(P, e), \\ \qquad Q' \in after(Q, e), e \in (A \cap B)_\surd\} \end{array} \right.
\end{aligned}
$$

Replicated

All of the above operators have a both a syntactic and value form. The step which reduces one to the other corresponds to the promotion of operators in the operational semantics. We use this same reduction from syntactic to value form to eliminate any replicated operations in favour of the binary equivalent. This is comparatively simple, since all the control expressions are required to be finite.

The replicated parallel operation is slightly more complex than the others since both communication sets and processes must be manipulated.

Prefix

The prefix operation is the most complicated of the process operators. The implementation needs to calculate the set of values offered by the prefixing. Given an element of that set, it also needs to bind it to the fields in order to extend the environment. The extended environment can then be used to evaluate the body of the prefix. The first two of these steps are driven by the structure of the fields which form part of the prefix operation.

Considering the calculation of the initials, we write

```
class vPick_ : public vProcBase
{
  Value c;
  FieldList fs;
  Syntax p;
  Env e;
public:
  Value inits() const
  {
    return vProduct(vDot, c, match(c.extensions(), fs, e).set());
  }
};
```

where `match` restricts the extensions of channel `c` to those permitted by the fields. It does this by decomposing the set of extensions as directed by the fields

```
Value match(const Value &s, const FieldList &fs, const Env &e)
{
  if(fs.null())
  {
    Value r = s.type().as_set();
    if(r != vGlue(vDot(), vEmpty())) offer_mismatch();
    return r;
  }
  else
  {
    Value x;
```

```
    if(fs.cdr().null())
      x = fs.car().match_final(s, e);
    else
      x = fs.car().match(s, e);
    Value r = vEmpty();
    for(VSet i = x.set(); !i.empty(); i = i.rest().set())
    {
      Value y = i.pick();
      Env f = e.add(fs.car().bind(y, e));
      r = vUnion(r,
            vProduct(vDot, y, match(y.drop_from(s),
                                    fs.cdr(), f).set()));
    }
    return r;
  }
}
```

If the list of fields is empty, then we check that the remaining type consists of simply the empty dotted value; if not, then we complain. If there are fields remaining, then the prefix of the type which the first field matches is selected (using a different method for the final field to allow trailing inputs to 'stretch' in the way described earlier.) For every value in this type, we then generate the matches of the remainder of the fields in an environment where the first field has been bound to the value selected. This is necessary for cases such as $c?x : \{0..5\}?y : \{x..5\} \rightarrow STOP$ where the subsequent matches are affected by the value just bound.

Given a value from the initials that this computes, this stepwise binding can be carried out again to yield an environment to be paired with the body of the prefix operation in order to calculate the afters.

8.13 Summary

Although constructed primarily as a reference implementation and 'proof-of-concept', the implementation outlined in this chapter formed the basis for the 'compiler' component in FDR2. That translates a CSP process description into a collection of low-level transitions systems and a tree of operators which combine those low-level systems to form the process.

The implementation is also used in Formal Systems' ProBE which enables processes to be explored, action by action. That combines the operational semantics described here with techniques for extracting compact descriptions of actions and

processes.

The main requirements for this implementation were simplicity and flexibility. Efficiency, such as that to be gained from less direct implementation techniques, was a secondary consideration. Nevertheless, the current FDR compiler is capable of enumerating state-machines at several hundred configurations per second.

The next chapter discusses possible extensions to the implementation.

Chapter 9

Adding extensions

This chapter proposes several new operators and other extensions, discussing in detail the mechanics of adding them to the language.

In general, extensions introduce the possibility of dialects into the language. Despite the long tradition of using CSP as a basis for experimentation, the addition of extensions to a machine-readable form of CSP conflicts directly with the need for tools to remain compatible with each other.

Nevertheless, there are some constructs whose addition to the language is demanded by a large enough proportion of the user-base to make their addition necessary; the growth of CSP provides enough evidence for this. Equally, the evaluation of proposed new extensions means that the mechanisms for incorporating them into tools (to see if they are as useful as claimed) must be documented.

9.1 Timeout

Timed CSP introduced a timeout operator, written $P \overset{t}{\rhd} Q$, into the language. The projection of the timeout operator into the *untimed* version of CSP turns out to be useful for some applications. The timed version offers P for a period of t time units, and if no initial action of P is chosen within the period, P is withdrawn and Q is offered instead. Informally, the loss of the timing information when we move to the untimed model means that it is no longer possible to predict the transfer; before the transfer we can choose either of the processes $(P \Box Q)$ and after the transfer we can choose only the latter (Q), but the transfer is invisible so we have the identity

$$P \rhd Q \quad = \quad (P \Box Q) \sqcap Q$$

Adding the operator to the language proceeds in several steps. First, we must choose an ASCII representation for the symbol and add it to the lexer.

```
{w}"[>"{w} { return TIMEOUT; }
```

Note that the representation chosen does not change the meaning of *any* existing scripts, since the two component symbols never occur adjacent to each other. Preserving the meaning of existing scripts should be a higher priority than the readability of the representation.

The precedence and associativity of the operator must be decided; we place it between the interrupt and semicolon operations and make it non-associative; since the timed version is not associative (without adjusting the timeout values) it seems simpler to make the untimed version non-associative also. This controls the positioning of the symbol in the parser source file.

```
%left INTR
%nonassoc TIMEOUT
%left SEMI
```

In the same file a reduction rule for the operator is added to the grammar.

```
_proc : proc TIMEOUT proc
             { $$ = sTimeout($1, $3); }
```

The class corresponding to the node in the syntax tree is defined in the file `s_proc.cpp`. The output functions are straightforward, and the value function can be simply written using the above identity; this avoids the need to define the corresponding class in the value hierarchy.

```
class sTimeout_ : public sBase
{
  Syntax p, q;
public:
  sTimeout_(const Syntax &x, const Syntax &y)
    : p(x), q(y)
  { }
  Value value(const Env &e) const
  {
    Value a = vClosure(p, e);
    Value b = vClosure(q, e);
    return vNdet(vBox(a, b), b);
```

```
  }
  ostream &output(ostream &os) const
  { return os << p << "[>" << q; }
};
```

The identity cannot be exploited at the syntactic level, avoiding the use of the syntax class, since tools which work on the syntax tree would then be unable to identify timeout operations. For example if the parser used

```
_proc : proc TIMEOUT proc
             { $$ = sNdet(sBox($1, $3), $3); }
```

then the timeout operator would be output as $(P \Box Q) \sqcap Q$, which is incorrect.

Finally, the class is hidden behind a construction function

```
Syntax sTimeout(const Syntax &x, const Syntax &y)
{ return new sTimeout_(x, y); }
```

Together with a declaration of the above function in the appropriate header file, this adds the operator to the framework using less than 25 lines of code.

9.2 Override

As well as the external choice operation $P \Box Q$, which chooses between arbitrary processes, the language presented in [18] had a simpler choice operator which offered a choice between a list of uniquely guarded processes $a \rightarrow P \mid b \rightarrow Q \mid c \rightarrow R$. Since it is always equivalent to the corresponding expression written using the external choice operation, and the n-ary nature of the operator complicates presentations of the semantics, the guarded-choice operator is no longer commonly mentioned in grammars for CSP (such as that at the beginning of Chapter 2).

The guarded-choice operator is not defined when guards are repeated, as in $a \rightarrow P \mid a \rightarrow Q$. The equivalent form is non-deterministic when written using the external choice operator, since

$$a \rightarrow P \Box a \rightarrow Q \quad = \quad a \rightarrow (P \sqcap Q)$$

The alternative to forbidding the repeated guards is to provide a variant of the guarded-choice operator which indicates the direction in which the ambiguity is to be resolved. This suggests the addition of two operators $+|$ and $|+$ such which bias

the selection in the case of ambiguity, but otherwise behave like guarded choice operator, so that

$$\begin{aligned} a \to P \,|+|\, a \to Q &= a \to P \\ a \to P \,|+\, a \to Q &= a \to Q \end{aligned}$$

and, for $a \neq b$,

$$a \to P \,|+|\, b \to Q = a \to P \,|+\, b \to Q = a \to P \,|\, b \to Q$$

Semantics

Unlike the timeout operation, which can be defined using a simple equivalence, the override operator requires new clauses in the semantics. We discuss only the '$|+$' of the operator here, since the operators can obviously be defined in terms of each other. The operator has a natural definition in the operational semantics

$$\frac{P \xrightarrow{\tau} P'}{P \,|+\, Q \xrightarrow{\tau} P' \,|+\, Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \,|+\, Q \xrightarrow{\tau} P \,|+\, Q'}$$

$$\frac{Q \xrightarrow{a} Q'}{P \,|+\, Q \xrightarrow{a} Q'} \qquad \frac{P \xrightarrow{a} P' \wedge Q \not\xrightarrow{a} Q'}{P \,|+\, Q \xrightarrow{a} P'}$$

Unfortunately, these definitions are not correct; the problem is caused by the $Q \not\xrightarrow{a} Q'$ clause in the final rule. This allows internal actions as part of Q to conceal actions which should override those of P. For example, using the definition above, if $R = a \to SKIP$

$$\begin{aligned} ((a \to STOP) \,|+|\, (R \sqcap R)) &\xrightarrow{a} STOP \\ ((a \to STOP) \,|+\, R) &\not\xrightarrow{a} STOP \end{aligned}$$

which is unfortunate, since $R \sqcap R = R$.

It is obvious that we require some property to hold of Q which prevents this problem. Clearly, requiring that Q be guarded is sufficient, if restrictive. More generally, we could require that Q may not perform any internal actions, but this requires the internal actions to be visible, which they are not in the denotational semantics.

In the case of the operational semantics, it is simplest to refuse to carry out an initial action of P if it is obvious that Q can perform the same action, or Q can perform an internal action so that it is uncertain whether Q can perform the same action. This requires changing the final clause to

$$\frac{P \xrightarrow{a} P' \wedge Q \not\xrightarrow{a} Q' \wedge Q \not\xrightarrow{\tau} Q'}{P \mid\!+ Q \xrightarrow{a} P'}$$

This is semantically sound, since either Q reaches a stable state after finitely many internal actions or it is divergent.

Implementation

The actual implementation of the operators is straight-forward. Again, the operators are added to the lexer.

```
{w}"|+"{w}  { return CHOOSERIGHT; }
{w}"+|"{w}  { return CHOOSELEFT; }
```

with parser rules

```
proc    : proc CHOOSELEFT proc
            { $$ = sChooseLeft($1, $3); }
        | proc CHOOSERIGHT proc
            { $$ = sChooseRight($1, $3); }
        ;
```

where both `sChooseLeft` and `sChooseRight` are implemented as simple binary operators in terms of

```
class vOverride_ : public vProcBase
{
  Value p, q;
public:
  Value inits() const
  {
    Value a = p.inits();
    Value b = q.inits();
    if(member(vTau(), b.set()))
      return b;
    else
      return vUnion(a, b);
  }
  Value after(const Value &e) const
  {
    if(is_tau(e))
```

```
      return vJoin(vOverRide, p.after(e).seq(), q.after(e).seq());
    else if(member(e, q.inits().set()))
      return q.after(e);
    else
      return vCat(p.after(e), q.after(e));
  }
};
```

9.3 Boolean guards

The ALT construct of occam corresponds to the choice operators of CSP, but it adds a feature not found in most CSP dialects: the branches of the alternative can have boolean guards. For example

```
ALT
  round & up ? x
    down ! x
  square & left ? x
    right ! x
```

so that the precise set of events offered by the choice is controlled by boolean guards. When modeling such a construct in CSP, there are two alternatives. The first, and most unwieldy, is to rewrite the occam so that the evaluation of the guards is carried out at an outer level and the guards can be removed from the choices. The choice then becomes

```
IF
  round AND square
    ALT
      up ? x
        down ! x
      left ? x
        right ! x
  round AND NOT square
    ... up
  square AND NOT round
    ... left
  NOT round AND NOT square
    ... STOP
```

which converts to CSP as the corresponding cascaded test. This approach is complex and scales extremely badly to larger problems.

A much better approach is given by noting that $STOP$ is the identity for the external choice operation, so that

$$\begin{aligned} P \,\Box\, STOP &= P \\ STOP \,\Box\, P &= P \end{aligned}$$

which allows us to write the above example as

$$\begin{array}{l} (\text{if } round \text{ then } up?x \rightarrow down!x \rightarrow SKIP \text{ else } STOP) \\ \qquad\qquad\qquad\qquad\qquad\qquad \Box \\ (\text{if } square \text{ then } left?x \rightarrow right!x \rightarrow SKIP \text{ else } STOP) \end{array}$$

This 'if b then P else $STOP$' term occurs when converting occam into CSP (see [34] for details), but it is also a useful idiom for specifications. It is also sufficiently bulky when written out in this way that a shorthand for it could prove useful. We thus introduce a new 'guard' operation $b \,\&\, P$ which is defined by the equivalence

$$b \,\&\, P = \text{if } b \text{ then } P \text{ else } STOP$$

Adding this to the language is a simple operation. After adding the pattern to the lexer

```
{w}"&"{w} { return GUARD; }
```

the token is defined in the grammar, giving the operator a precedence between that of sequential composition and prefixing.

```
%left SEMI
%nonassoc GUARD
%right ARROW QUERY PLING
```

and a reduction rule is added

```
_proc : bool GUARD proc
             { $$ =  sGuard($1, $3); }
```

Finally, the class for the syntax node simply uses the syntactic equivalence to provide a direct implemenation for the operator.

```
class sGuard_ : public sBase
{
  Syntax b, p;
public:
  sGuard_(const Syntax &x, const Syntax &y)
    : b(x), p(y)
  { }
  Value value(const Env &e) const
  { return b.value(e).logic().cond(vClosure(p, e), vStop()); }
  ostream &output(ostream &os) const
  { return os << b << "&" << p; }
};
```

9.4 Transparent operations

The FDR tool compiles CSP processes into finite state machines, which it then compares to check the refinement relation. The second generation of FDR uses a variety of representations for state machines and there are a number of operations which convert between these representations; typically the representation is chosen in an attempt to minimise the space taken by the process.

While such 'compression' operations should be hidden from the majority of users, it is useful to have an explicit form for the operations in the syntax, since this allows easy experimentation during the evaluation of the operations and is likely to be useful for advanced users working on large problems. The group working on FDR thus requested a way of expressing compression operations on processes within the syntax.

The most reasonable form for the compression operators is as prefix functions. Thus, we expect

$$P \quad = \quad compress2(compress1(Q) \underset{A}{\|} R)$$

to selectively compress components of P as it is compiled. The FDR compiler could then bind these compression operations to the corresponding actions, and this would be sufficient to allow compression in FDR. Unfortunately, scripts intended for FDR would then be incompatible with every other tool.

If the new operations actually extended the language, then this incompatibility between FDR and other tools *might* be acceptable. However, the compression operations have an important property: they do not change the processes in any way, they simply optimise the underlying representation. This implies that any tool

should be able to understand the above fragment provided it ignores the compression operations.

One quick solution simply marks off a range of identifiers, such as those beginning with 'compress,' as special operations which tools are free to ignore. This is unsatisfactory for two reasons. The first is that this fails to support any similar requirements for other tools in the future; this is likely to end in a range of 'magic' prefixes each of which will require modifications to existing tools as new requirements are identified. The second is much simpler: the compressions are not simply numbered (as the example might suggest), they can be given meaningful names like 'normalise' and 'diamond'; combining these with a prefix results in extremely long identifiers.

If it is not possible to recognise these special operations by name, then there must be an explicit mechanism in the language for identifying them. This suggests that scripts can define these operations using some special syntax; tools are free to supply 'real' definitions for those operations which they understand.

For example, an FDR fragment might be

```
transparent diamond, normal
P = diamond(normal(Q) [| A |] R)
```

where the transparent definition binds the identifiers to the identity function *unless the tool has a built-in interpretation which it may activate.* Thus tools which understand these operations thus use the built-in binding for these identifiers, while all other tools treat this as

```
diamond(x) = x
normal(x) = x
P = diamond(normal(Q) [| A |] R)
```

(It is important that the built-in binding is activated only when the transparent definition is seen, since otherwise scripts from tools which support such built-ins will tend to omit the transparent declarations.)

Implementation

The first stage in the implementation is, as usual, to add a rule for the token to the lexer

```
"transparent" { return TRANSPARENT; }
```

The token is the defined in the grammar

```
%token CHANNEL DATATYPE TRANSPARENT
```

and a reduction rule is added

```
defn    : TRANSPARENT names
                { $$ = dTran($2); }
        ;
```

which constructs a definition class

```
class dTran_ : public dBase
{
  SyntaxList names;
public:
  dTran_(const SyntaxList &x)
    : names(x)
  { }
  ostream &output(ostream &os) const
  { return os << "transparent " << names; }
};
```

For transparent definitions, all the identifiers are bound to the identity function unless a built-in definition is available. If we assume that `lookup` returns either the identity or the built-in as appropriate, we have

```
PreEnv dTran_::change() const
{
  PreEnv res;
  for(SyntaxList i = names; !i.null(); i = i.cdr())
    res = res.add(lookup(i.car());
  return res;
}
```

The `lookup` operation can be implemented using a repository mapping names to definitions. The repository is built by the tool during initialisation and searched as part of the `lookup` operation.

9.5 Assertions

During testing of the operational semantics, it was extremely useful to have a mechanism which allowed 'regression tests' to be embedded within scripts. These consisted of marked logical expressions, which tools could automatically locate and verify. It became apparent that such a mechanism could be useful for locating general problems with scripts, so the mechanism was cleaned up and is documented here.

The checks to be performed are placed in special 'definitions' at the top level of a script. An additional method is then provided in the definition base class which checks the validity of the assertion. All 'normal' definitions simply ignore this method, but the assertions reduce the boolean value which they contain and indicate the result.

The extension to the grammar is simple

```
defn    : ASSERT bool
                { $$ = dAssert($2); }
```

where the lexer has

```
"assert"        { return ASSERT; }
```

and the basic implementation is

```
class dAssert_ : public dBase
{
  Syntax b;
public:
  int check(const Env &e) const
  {
    int okay = b.value(e).logic().cond(1, 0);
    cerr << (okay ? "Okay" : "Fail") << ": " << b << endl;
    return okay;
  }
};
```

A tool can then be constructed simply which parses a script and forces the `check` method to be called for each definition which results.

9.6 Refinement

The main application of this enriched form of CSP thus far has been to support refinement checkers. When using such tools, the checks to be performed are indicated by choosing a pairs of processes from a list of those available, then selecting the precise refinement relation. This has the unfortunate effect that it may be unclear which processes in a script should be compared. It has also inspired the construction of simple batch languages to allow large numbers of checks to be performed without requiring user interaction.

It seems much more desirable to include the checks within the script itself, since this provides a useful form of documentation and also removes the need for an external batch file. However, the refinement relation is *not* part of the expression language; that would break the clean separation between expressions and processes and allow nonsensical definitions like

$$P \;=\; \text{if } P \sqsubseteq STOP \text{ then } SKIP \text{ else } STOP$$

Refinement tests depend upon processes in the same way that processes depend upon expressions: they form another layer in the model. The most natural place to make use of these tests is in the assertions described in Section 9.5. While it may be useful to allow logical connectives to be used with these expressions, this option can be left as a future extension.

Currently, assertions simply take a boolean value, as in

```
defn    : ASSERT bool
                { $$ = dAssert($2); }
        ;
```

The first step is to change this to a production which includes booleans and refinement checks

```
check   : bool
                { $$ = $1; }
        | proc refined proc
                { $$ = sRefine($1, $2, $3); }
        ;

defn    : ASSERT check
                { $$ = dAssert($2); }
```

Since there is a refinement relation for each of the standard models of CSP, we cannot use a single symbol for the refinement relation. The convention adopted

within FDR will be used, where refinement is indicated using `[=` and the precise model is given by letters inside the symbol. Failures refinement uses `[F=`, traces refinement uses `[T=`, and failure-divergences refinement uses `[FD=`. The parser accepts any string of upper-case letters inside the relation; tools are expected to complain about any which they do not recognise and attempt to continue. Since the only way a refinement check can be used is directly in an assertion

```
assert P [T= Q
```

an acceptable way of continuing is to return true when evaluating unrecognised refinement relations. If logical connectives are added around assertions then a more general mechanism will be required, probably involving exceptions.

The lexer recognises refinements using

```
uppers                  [A-Z]*

{w}"["{uppers}"="{w} { return REFINED; }
```

and the parser then uses

```
refined : REFINED
                { $$ = Text(yytext); }
        ;
```

The default implementation recognises *no* refinement relations, so we have

```
class sRefine_ : public sBase
{
  Syntax lhs;
  Text text;
  Syntax rhs;
public:
  sRefine_(const Syntax &x, const Text &y, const Syntax &z)
    : lhs(x), text(extract(y)), rhs(z)
  { }
  Value value(const Env &e) const
  {
    Winge("Refinement checks not yet implemented.");
    return 1;
  }
};
```

where `extract` pulls the text indicating the refinement relation out from the symbol.

Coding refinement checks directly using the operations provided by the process classes is possible, but it requires a mechanism for spotting when pairs of processes have been previously checked for refinement. This requires an equality test on processes, the natural candidate for which is simple structural equality on the value involved. This in turn has a natural implementation on terms of dynamic-casting, which is still not widely available in C++ compilers.

9.7 Summary

This chapter has discussed the addition of several extensions to the language presented in Chapter 2, including the effort involved in supporting them as part of the implementation of Chapter 8.

In contrast, the next chapter discusses some of the differences between the language presented here and some of the intermediate versions, such as that presented in [33].

Chapter 10

History

During the development of the language presented here, some features have been added, more have been modified and a few have been rendered obsolete. This chapter discusses some of these features, especially those which may be familiar to users of FDR1.

10.1 Channels

The version of the language described in [33] made no use of channel declarations: it was assumed that the type of channels could be deduced by the system. While this was acceptable in general, FDR1 encodes events into its own internal representation when they are encountered, and it obviously requires that the set of all such events be finite. The easiest way to ensure this is to make the protocol associated with each channel finite.

Unfortunately, the types which can be deduced for a channel are rarely finite: constraining a protocol from a channel of *Int* to a channel of $\{1..6\}$ requires sophisticated analysis, and may not be possible in some cases.

Scripts which were to be used with FDR1 thus required channels to be declared (and to be of finite size). A mechanism was needed to allow this, while permitting other tools to ignore the declarations, and the pragma mechanism from ANSI-C was taken as the basic model. Pragmas were to be used for extensions to the language which tools were permitted to ignore if they did not understand the particular extension being used. In the case of channels, this led to declarations of the form

```
pragma channel throw : {1..6}
```

The implementation simply stored the entire line after the pragma keyword as text

and tools were expected to parse the body of the pragma themselves, ignoring the pragma if they did not recognise the first word of the pragma. In the case of channels, this meant that FDR1 was responsible for parsing and manipulating the expressions which formed the protocol of the channel, restricting the expressions which could be used to simple names and set expressions.

In the light of experience with FDR1 users, two facts became apparent. The first is that channel declarations are extremely useful: they provide important documentation on the intended use of a channel and allow the tool to report abuses, even in the absence of a static type-checker. The second is that the restricted expression language allowed in the declaration was hard to justify and seemed likely to become more of a problem as datatypes were added and protocols became more sophisticated.

The natural conclusion was that channel declarations should become part of the core language, rather than an FDR-specific extension. This extends the benefits of declared channels to a wider base of users and also allows the use of the full expression syntax in the declaration. Thus, the 'pragma channel' mechanism of early version of the language was replaced by the 'channel' mechanism described in earlier chapters.

10.2 Standard ML

The compiler implemented as part of FDR1 was built using Standard ML. It had limitations on the use of parameters, primarily to simplify the implementation. Only 'low-level' sequential processes could be parameterised; parallel processes and simple values were not permitted to have arguments. This made it impossible to define auxiliary functions within the script.

Since pragmas provided a mechanism for adding arbitrary extensions, they were used to allow scripts to access ML defined functions. Thus

```
pragma opaque "ML" sort
```

indicated that a function was defined in the ML environment which was part of the FDR1 compiler, and that it could be used from within the CSP script.

To avoid the complication of loading extra ML definitions into the FDR1 system, another pragma was implemented which allowed the ML to be embedded within the CSP script. The '`inline "ML"`' pragma has been used on perhaps a handful of occasions.

10.3 Use of pragmas

Since both of the uses of pragmas outlined above are no longer required in the current language, pragmas should be considered obsolete. They are still present in the parser and lexer since they provide a convenient mechanism for prototyping simple language extensions, but their use should be avoided in 'production' software. For example, the transparent mechanism of 9.4 was prototyped using pragmas but is now implemented directly as a language feature.

10.4 Input patterns

The handling of patterns in inputs and outputs has changed subtly from that used in [33]. In that document, dots in communications repeated the direction of the last communication, so that $c?x.y$ and $c?x?y$ were considered the same, as were $c!x.0$ and $c!x!0$. In this document, the first of these examples is actually implemented as $c?(x.y)$ inputting a value and then decomposing it using a dot-pattern. The main result of this change is that $c?x : y.z$ is no longer considered acceptable; it previously meant the same as $c?x : y?z$.

10.5 Summary

This chapter has discussed some of the changes which have been made to the machine-readable form as it has developed. The next will explore changes which might be made in the future.

Chapter 11

Extensions

We now consider possible extensions to the work presented here. These can be broken down into three major categories.

First, and simplest, there are those changes which correspond to incremental improvements to the implementation given in Chapter 8, but require no major changes to the language or the semantic models.

Secondly, there are changes which add extra syntax, or relax some of the restrictions in the language with a corresponding change to the implementation, but which can make use of the standard CSP models and the semantics given here.

Finally, some of the many timed or probabilistic versions of the language can be incorporated, replacing the standard failures-divergences model used here with another known process model, with fairly complex follow-on effects on the language, semantics, and implementation.

11.1 Improvements

Static type-checking

The language presented has been carefully designed to permit the implementation of a static type-checker. The basic approach, as indicated in [33], can be adapted from that given in [27] with unification on type-signatures. The language presented in [33] differed in two significant respects from that given here.

Firstly, channels were not explicitly declared and this led to a notion of the interface of a process. Given a language with explicit channel declarations, such interfaces are no longer required and the type-checker can be simplified to an approach closer

to that of [27], provided channel declarations are restricted to the top-level of the script.

Secondly, the earlier version of the language omitted datatypes. These complicate the checker since it must incorporate rules to handle the dots used to build datatypes (and channels). These rules are similar to those required for function application, but must be modified to cope with the associativity of the dot operation.

Given the existence of a type-checker, the *error* clause prominent in all the semantic models can be eliminated, since a primary requirement of a static type-checker should be that all correctly typed programs cannot encounter run-time type errors.

Full pattern-matching

Pattern-matching, as used in functional languages such as Gofer, comes in two distinct forms. The first form uses patterns to decompose a composite value without the need to apply the selector functions. For example, in Gofer, the function which takes a pair and adds the two components can be written as either

```
sum p = fst p + snd p
```

or, avoiding the use of *fst* and *snd* as

```
sum (x,y) = x + y
```

The second form uses repeated definition of a name on different branches of a type. For example, again in Gofer, we can write

```
null []     = true
null (x:xs) = false
```

The language presented here has limited support for pattern-matching in decomposition, in that composites built from tuples and dots may be decomposed as part of function application. Thus, the first example translates directly into CSP as

$$sum((x, y)) \quad = \quad x + y$$

However, decomposition is not supported in definition clauses, so that

$$(x, y) \quad = \quad (2, 3)$$

will report a 'bad binding attempt.' The two cases are treated differently, since in the first case the binding is non-recursive (and thus made to a value) while in the second we need to bind to syntax in order to form the recursion with any other simultaneous definitions. When binding to a value it is easy to decompose the value to match the pattern, but decomposing the syntax requires a more sophisticated approach. The obvious approach is to apply the selectors to the syntactic term, so that the above definition is transformed to

$$\begin{array}{lcl} x & = & \mathit{fst}((2,3)) \\ y & = & \mathit{snd}((2,3)) \end{array}$$

While this works, it is inefficient, since the right-hand-side of the definition is copied and this can lead to the duplication of significant computation. This can be avoided by applying the selectors to a shared copy of the value after the recursion has been unrolled; this involves the addition of an extra binding phase in the code which calculates the change in an environment caused by a set of recursive definitions.

Adding support for two other common pattern-matching mechanisms is simpler. The 'discard' pattern is used when the value which would otherwise be bound is unimportant, as in

$$\mathit{fst}((x,_)) \quad = \quad x$$

This can be supported simply by making '$_$' a syntactic term which can be bound to any value, but which produces no change in the environment as a result. Similarly, the 'as' pattern allows both the whole and parts of an expression to be bound at the same time, as in

$$\mathit{order}(p@(x,y)) \quad = \quad \text{if } x \leqslant y \text{ then } p \text{ else } (y,x)$$

Again, this can be handled by adding a special syntactic class whose effect on binding is the sum (checked to be disjoint) of the binding caused by its two components.

The second form of pattern-matching is not supported at all by the language described so far. The main problem is that each value in a block of definitions is assumed to be defined precisely once; duplicate definitions are considered to be an error. Adding support for branched definitions thus requires that this case is recognised and the definitions are combined in some way. The natural way to do this is to combine the various terms into a 'branched function' value which is built as a series of lambda terms which are to be applied until one matches the argument supplied. This in turn requires that a lambda term can indicate whether it finds a supplied argument acceptable. The corresponding simple and branched lambda classes are roughly of the form

```
class vLambdaBase_ : public vNormal
{
public:
  // Provide down-cast
  VLamdba lambda() const;
  // Indicate if the argument is acceptable
  virtual int accepts(const Value &) const = 0;
};

class vLambdaSimple_ : public vLambdaBase_
{
  // As current lambda term, with the addition of
public:
  int accepts(const Value &) const;
};

class vLambdaBranch_ : public vLambdaBase_
{
  List<VLambda> branch;
public:
  int accepts(const Value &x) const
  { return 1; }
  // Find the first element of branch which
  // accepts the value, then apply it.
  // If none, the function was not total so
  // indicate an error.
  Value apply(const Value &) const;
};
```

This approach delegates the testing of each alternative to the individual branches and is inefficient; it is common for the decision to be optimised in functional languages by testing the structure of the value supplied directly. However, this does indicate that branching pattern-matching can be supported with comparatively little modification to the implementation described in Chapter 8.

Given that such an implementation is possible, we now consider how to support pattern-matching on sequences; these play the part taken by lists in most functional language, but the history of CSP means that, while lists support the simple patterns corresponding to the empty list and a cons cell, CSP users expect sequence pattern-

matching to allow examples such as

$$\begin{aligned} f(\langle\rangle) &= 0 \\ f(\langle a\rangle) &= 1 \\ f(\langle a, b\rangle) &= 2 \\ f(\langle a, b, c\rangle) &= 3 \\ f(\langle a, b\rangle^\frown s^\frown \langle d, e\rangle) &= 4 + f(s) \end{aligned}$$

which of course is intended to calculate the length of a sequence.

Supporting this type of pattern-matching on sequences requires a notion of when a sequence has a definite length. In the example above, and in general, sequence literals have a definite length while simple names do not. We will call those patterns which do not have a definite length 'sloppy', and those which do 'concrete'. A sequence pattern can be guaranteed to have at most one match against an argument if it is a catenation of simple sequence patterns, at most one of which is sloppy.

Given such a uniquely-matching sequence pattern which has a sloppy component, it will accept a value if the length of the value equals or exceeds the total length of the concrete patterns. Even more simply, a sequence pattern match which is totally concrete accepts when the length of the argument is the same as the total length of the pattern. (This assumes that the component simple patterns are always accepting.)

The implementation of the binding of such patterns simply removes the concrete patterns one at a time, binding them to either an initial of final subsequence of the argument before binding any remaining sloppy pattern (if any) to the remainder of the argument.

Type parameters

The language presented so far is more restrictive in the datatypes which can be defined than normal functional languages such as Gofer. The restriction appears most directly in the grammar clause which defines

```
defn    : DATATYPE name EQUAL vartype
        ;
```

which means that it is not possible for user-defined types to take arguments, unlike the supplied set and sequence types.

Relaxing this restriction should be as simple as extending the grammar, binding the name of the type to a lambda term if the type is parameterised. The domain

construction and semantic equations given in Chapter 3 should remain valid. However, this has not been investigated and the change also affects the construction of a static type-checker.

Recursive types are supported by the language presented here, but their use is problematic with tools (such as FDR) where channels are required to have finite types.

Local channels

The language presented here restricts channel and type-definitions to the top level of a script. Since definitions are localised using let clauses which tend to be comparatively small, this is not a serious problem, although local channels would be useful in some cases, the chaining operation described in Section 2.12 being an example.

Although fragments such as

$$P \quad = \quad \left\{ \begin{array}{l} \text{let} \\ \quad \text{channel } c : Int \\ \quad Q = c!0 \rightarrow STOP \\ \text{within } c?x \rightarrow Q \end{array} \right.$$

and

$$S \quad = \quad \left\{ \begin{array}{l} \text{let} \\ \quad \text{channel } c : \{0..3\} \\ \text{within } \{|c|\} \end{array} \right.$$

seem plausible, it is not clear that the 'exposed' values built from a tag which is no longer accessible are well-behaved, in either the semantics or the implementation. In particular, given S above, it seems likely that the implementation will permit

$$\begin{array}{l} \text{channel } c : \{1, 2\} \\ b = member(c.1, S) \end{array}$$

since it fails to distinguish the different instances of c. This may be as simple as marking the tags to make them unique, but for now this issue is avoided by restricting type definitions to the top-level of scripts. If a mechanism is provided for more substantial local definitions, for example the module mechanism outlined in subsection 11.2, this restriction may become unacceptable.

Pure operational semantics

The semantics of [31] and [7] are implemented purely in terms of a transition relation, while that of Chapter 4 makes extensive use of a denotational semantics for the expression language. It should be possible to produce an alternative semantics where this expression evaluation was also carried out using transitions, labelled with another internal event on the reduction steps. The congruence between that (fully) operational approach and the purely denotational could be demonstrated by an extension of the approach used in Chapter 6.

11.2 Additions

Scaling

The only mechanism described so far for localising identifiers is the `let` clause. This restricts the scope of the definitions to a single expression; the body of the clause. It seems likely that as larger CSP scripts are constructed, this may prove to be inadequate.

For example, it may be that some definitions are used several times, but still have a restricted scope; essentially this requires restricting the scope of some definitions to another group of definitions rather than a single expression. This corresponds to the use of modules in most programming languages; for example, C typically implements a module as a single file exporting some definitions, but restricting some to file scope.

This suggests a solution to the problem along the lines of

```
module
  f(x)    = x + 1
  g(x, y) = f(x * y)
  a       = g(0,0)
  b       = g(a,1)
exporting g, b
```

which allows teams to work on separate parts of a system, producing well-defined exports without cluttering a global name-space.

Of course, given suitably powerful pattern-matching, the above construct can be regarded as syntactic-sugar for

```
(g,b) = let
```

```
  f(x)    = x + 1
  g(x, y) = f(x * y)
  a       = g(0,0)
  b       = g(a,1)
within (g, b)
```

although this is less convenient when many identifiers are exported.

As observed in subsection 11.1, if scripts reach the level of complexity where modules are necessary, local channels are likely to be required as well.

Events

Functions may not be communicated. Allowing the communication of expression-valued functions is simple within the framework used here, yielding a richer model. However, it is less operationally intuitive, and would introduce serious difficulties when implementing the semantics. Allowing the communication of process-valued functions would be equivalent to allowing the communication of processes; this is discussed below.

Processes may not be communicated. This is a standard CSP restriction, and removing it means that none of the standard models of CSP are useful. Essentially this means that we can no longer build $\mathcal{N}$ when F is complete, but must solve the (now mutually recursive) domain equations for $\mathcal{N}$ and F simultaneously. Alas, the standard formulation of $\mathcal{N}$ involves powersets rather than domain constructors.

Replicators

The language presented here restricts the controlling expression of replicated operators to be finite sets or sequences as appropriate and limits the hiding and renaming operations to preserve finite non-determinism.

Removing the finite non-determinism restriction means moving to the definedness order of [31]. The proofs are complicated by the fact that the operators can no longer be unrolled. Although theoretically relaxing the restriction is possible, the problems it introduces into any implementation would seem to argue against it.

Channels

As presented here, events are constructed from the 'anchor' of a named channel. However, some versions of CSP allow the use of simple values as events. For

example,

$$\begin{aligned} P &= !7 \rightarrow SKIP \\ Q &= ?x \rightarrow SKIP \end{aligned}$$

defines two components communicating an *Int* without using a channel. This approach worked well in the alphabetised universe, but with parameterised parallel operators, the communication sets are simple sets. Effectively this involves widening Σ from the anchored part of $F^{\bullet}$ to all of it. In practice, it is not clear that this is a good idea; the cost of declaring channels is minor in comparison with the benefits in terms of type-safety, which this appears to discard.

The use of such unadorned events is related to the use of the 'prefix relabeling operator' which renames all the events of a process by prefixing them with another tag. For example, given

$$P = in?x \rightarrow out!x \rightarrow P$$

we might form $foo : P$ where

$$foo : P = foo.in?x \rightarrow foo.out!x \rightarrow foo : P$$

This operator is not compatible with the type system used here. If the communications in P are well-defined, so that *in* and *out* are, say, channels of *Int*, then it is not clear that the communications in $foo : P$ can be well-typed. For this reason, this operator is not supported by the language presented here. Although it provides a compact notation for this style of renaming, it is possible to achieve a similar effect using the renaming operation provided.

11.3 Richer models

Assignment

The addition of assignment to the language, while not in the spirit of CSP, is certainly a popular extension and adding it here requires the usual addition of a store to the semantic equations. This yields a semantics which (perhaps unsurprisingly) is similar to that presented in [16], although somewhat simplified by the fact that the refusal sets are the natural CSP ones rather than those required for the 'all or nothing' semantics of occam 2 inputs.

Time

Timed CSP can be regarded as an extension of ordinary CSP, adding some timed operators to those already present. In fact, some of the timed operators are intended to *replace* those already present, for example the timed transfer operator $\stackrel{t}{\longrightarrow}$ is intended to replace the untimed version, but we can regard $\rightarrow$ as a timed transfer which uses the default period of δ, so that

$$a \rightarrow P \quad = \quad a \stackrel{\delta}{\longrightarrow} P$$

Since the times used with the new operators are real numbers, the first stage in supporting Timed CSP must be to add a representation for real numbers. Unfortunately, the obvious notation of 1.2 is already in use as the dotted composite of two integer values; and this use is likely to be more common than the use of real numbers for timed operators.

We circumvent this problem by adapting the exponential notation common in other languages which adds an exponent to a floating point number using an 'e' character, as in $0.1e3$. We allow the numeric exponent to be omitted, in which case the 'e' may also be written as an 'f', so the above example can be written as $1.2f$ or $1.2e0$.

The timed operators take their time arguments as superscripts in the typeset form of the notation. As with the subscripts used for communication sets, this is impractical in an ASCII notation and an alternative representation would need to be found.

Probability and priority

There are several versions of CSP which incorporate probability and priority information, and little agreement on the operators required for these dialects. The question of a precise representation for these operators is therefore deferred until there is significant experience with the use of the languages on real examples. For further details, see Lowe's work on an experimental probabilistic refinement checker in [25].

Chapter 12

Conclusions

12.1 Summary

By providing a concrete language for the expression components of the CSP grammar, the work presented here has taken CSP from a blackboard notation to a language which is suitable for use with tools.

The expansion of the language has been accompanied by generalisations of the denotational and operational semantics. These combine a common expression semantics with each of the traditional denotational and operational semantics. The two semantics which resulted have been shown to be congruent using an extension of an existing congruence proof. Because of the clean separation between the expression and process semantics, this congruence clearly holds for most reasonable expression languages.

After laying this theoretical foundation for the language, both a parser and a reference implementation of the operational semantics have been described. These provide a suitable base for tool implementors.

Later chapters have discussed the work involved in adding new features to the implementation, and the impact of a range of changes on both the implementation and theory.

12.2 Tools

This work has made possible a number of tools. The brief summary given here cannot be comprehensive, but gives some indication of the range available.

FDR This was the first available CSP tool to build upon this work. It provides an industrial-strength explicit model-checker and uses CSP as its primary input notation. Early versions used the parser and an ML implementation of the operational semantics; current releases are based on the C++ implementation described here.

ARC The Adelaide Refinement Checker [39] uses the parser as the basis for another CSP model-checker using an alternative approach based on BDDs.

ProBE This CSP animator is built on the reference implementation of the semantics and allows users to explore CSP processes interactively.

Deadlock Checker Jeremy Martin's Deadlock Checker [26] uses FDR to decompose process descriptions into networks which can then be checked for properties which can guarantee deadlock. This allows deadlock-freedom to be established for systems far larger than could be handled using model-checking techniques.

Casper Gavin Lowe's Casper [24] automatically generates machine-readable CSP from concise descriptions of security protocols. FDR can then be used to automatically search for flaws in the protocols.

Wright Work at CMU (described in [1] and [2]) has been investigating a language for describing system architectures. This work has used CSP, both as a part of their language and to provide a semantics for the language as a whole. The Wright tool automatically generates CSP from their descriptions, allowing them to use FDR to check for required properties.

Concurrency Workbench Attempts have been made to use CSP as an alternative input notation to the Concurrency Workbench. These efforts are documented in [11]. The language used there is clearly based on the form of machine-readable CSP used by FDR1.

12.3 Applications

The work of Lazić on data-independence was initially inspired by the existence of machine-readable CSP, and the language used in [22] is still clearly related to that presented here.

This proliferation of tools has lead to applications of CSP in many areas. As my supervisor notes in [30], "it is only really since the advent of tools that CSP has

been used to a significant extent for the development and analysis of practical and industrial-scale examples."

Perhaps the most obvious area for CSP applications is that of network protocols. Work in this area includes [29] and [12].

More surprisingly, CSP has also been used to uncover flaws in a number of security protocols, encoding security checks as refinement properties. An example can be found in [23] and an extensive treatment of this approach in [38].

Related work has used an encoding of non-interference as determinism to address issues in railway-signalling (see [32] and [35]).

Work at the Bremen Institute for Safe Systems has used CSP as one of their tools in the pragmatic application of formal-methods to real-world applications. Some of the results are described in [8] and [9].

Most of these applications have involved tool-assisted analysis of CSP models. Many of them would have been impractical without the existence of CSP tools.

12.4 Acknowledgments

The work described in this document was funded by the U.S. Office of Naval Research under contract N00014–87–J1242.

None of this work would have been possible without the many and varied foundations laid by the CSP group of the Programming Research Group (PRG) at Oxford, including G.M. Reed, J. Davies, S.A. Schneider, and my supervisor A.W. Roscoe.

The assistance and advice of those developing tools based on this work has been invaluable, including D.M. Jackson and M.H. Goldsmith at Formal Systems, whose tool FDR helped drive the parser development, and G. Lowe at PRG whose prototype probabilistic refinement tool helped raise the issues of CSP dialects and ML support. See [14] and [25] for details.

Comments from J. Davies, R. Forster, M.H. Goldsmith, J.R. Hulance, D.M. Jackson, J.N. Reed, and C. Studholme on draft copies of this thesis were helpful and much appreciated.

Inspiration and notation was drawn from diverse corners. Work carried out while the author was working for Formal Systems on occam tools using the Inmos parse-tree inspired much of the framework for the parse-tree used here, and indeed the notion of such CSP tools. The existence of the Concurrency Workbench (a tool for the CCS-process algebra, described in [10]) also helped to motivate the development of tools for CSP.

M.P. Jones's reliable and portable Haskell subset, Gofer, reawakened my interest in functional languages, provided an excellent test bed for ideas, and influenced the script style.

The type system of occam 2 [19] and G. Barrett's proposal for occam 3 [5] had a strong influence on the way the event space Σ is constructed. The connection between datatypes and variant protocols grew out of comments by M.H. Goldsmith, which were in turn based on Barrett's work. The denotational semantics for occam 2 presented in [16] by B.G.O. Scott (together with M.H. Goldsmith and A.W. Roscoe) addresses similar problems, as it adapts the CSP models for use with a language with a more concrete event structure.

The need for direct language support for variant protocols was emphasised by work done in conjunction with K. Seidel on the connection between occam 2 and CSP in [34].

Appendix A

Theory

The results in this appendix are precisely those given in [7]. They are reproduced here since that paper remains unpublished. The results show that the space of synchronisation trees T used in Chapter 5 exists and has a complete metric which is sufficient to guarantee both the existence of the operators defined in Section 5.1, and of fixed points for equations defined using those operators.

A.1 Existence

Let $(S, \Theta, \rightarrow)$ be a finite-branching labelled transition system, as in Chapter 6, so that Θ contains Σ and a finite number of additional events.

A *skeleton* is a nonempty set $L \subseteq (\Theta \times \omega)^*$ such that, for $x \in \Theta, i, j < \omega$ and $u, v \in (\Theta \times \omega)^*$

$$u \in L \wedge v \leqslant u \;\Rightarrow\; v \in L \tag{A.1}$$

$$u^\frown\langle(x,i)\rangle \in L \wedge j < i \;\Rightarrow\; u^\frown\langle(x,j)\rangle \in L \tag{A.2}$$

$$u \in L \wedge y \in \Theta \;\Rightarrow\; \{k \mid u^\frown\langle(y,k)\rangle \in L\} \text{ is finite} \tag{A.3}$$

Let $\mathbf{S}$ denote the set of all skeletons. For $L \in \mathbf{S}$ and $x \in \Theta$, define

$$\begin{aligned} \lambda_L(x) &= \{i \mid \langle(x,i)\rangle \in L\} \\ init(L) &= \{x \mid \exists\, i \bullet \langle(x,i)\rangle \in L\} \end{aligned}$$

then by the definition of $\mathbf{S}$, $\lambda_L(x) = \{0..m-1\}$ for some $m \geqslant 0$. For $x \in init(L)$ and $i < \lambda_L(x)$ define

$$L \;after\; (x,i) \;=\; \{u \mid \langle(x,i)\rangle^\frown u \in L\}$$

then L *after* $(x,i) \in \mathbf{S}$.

For $L \in \mathbf{S}$ and $n < \omega$ we define a truncation operation

$$L \downarrow n \;=\; \{u \in L \mid \#u < n\} \cup \{u^\frown\langle(\tau,0)\rangle^k \mid u \in L \wedge \#u = n \wedge k < \omega\}$$

We also define prefix and sum operations on $\mathbf{S}$

$$\begin{aligned} x.L &= \{\langle\rangle\} \cup \{\langle(x,0)\rangle^\frown u \mid u \in L\} \\ L + L' &= L \cup \{\langle(x, 1 + max\ \lambda_L(x) + i)\rangle^\frown u \mid \langle(x,i)\rangle^\frown u \in L\} \\ \Sigma_{j<0} &= \{\langle\rangle\} \\ \Sigma_{j<n+1} L_j &= (\Sigma_{j<n} L_j) + L_n \end{aligned}$$

Combining these we define, for $i \in I, L_i \in \mathbf{S}, x_i \in \Theta$ and such that $\forall\, x \in \Theta \bullet \{i \in I \mid x_i = x\}$ is finite

$$\Sigma_{i\in I} x_i.L_i \;=\; \bigcup_{x\in\Theta} \Sigma_{j<n(x)} x_{i_j}.L_{i_j}$$

where $\langle i_0, \ldots i_{n(x)}\rangle$ is an enumeration of $\{i \in I \mid x_i = x\}$.

Now define an equivalence relation $\sim$ on $\mathbf{S}$ by $\sim = \bigcap_{n<w} \sim_n$ where

$$\begin{aligned} \sim_0 &= \mathbf{S} \times \mathbf{S} \\ L \sim_{n+1} L' &\Leftrightarrow \left\{ \begin{array}{l} \forall\, x \in \Theta \bullet \lambda_L(x) = \lambda_{L'}(x) \\ \wedge \\ \forall\, x \in init(L) \bullet \exists\, \pi \in \lambda_L(x) \leftrightarrow \lambda_{L'}(x) \bullet \\ \quad \forall\, i \in \lambda_L(x) \bullet L\ after\ (x,i) \sim_n L\ after\ (x, \pi(i)) \end{array} \right. \end{aligned}$$

where $A \leftrightarrow B$ represents the set of bijections from A to B. $\sim$ is an equivalence relation on $\mathbf{S}$ because of the finite-branching property. Let $[\cdot]$ denote the projection mapping and T be the set of equivalence classes. The elements of T are called *synchronisation trees*.

Since $\sim$ is a congruence relation with respect to the prefix, sum, and cut operations we may define these operations over T by defining

$$\begin{aligned} x.[L] &= [x.L] \\ [L] + [L'] &= [L + L'] \\ \Sigma_{i\in I} x_i.[L_i] &= [\Sigma_{i\in I} x_i.L_i] \\ [L] \downarrow n &= [L \downarrow n] \end{aligned}$$

A.2 Metric

Define $d_T : T \times T \to \mathbb{R}$ by

$$d_T(t,u) \quad = \quad inf\{2^{-n} \mid n \in \omega \wedge t \downarrow n = u \downarrow n\}$$

then d_T is trivially a metric. Note that if $t, u \in T$ and $n < \omega$ with $t \downarrow n = u \downarrow n$ then we can show that $\forall\, L \in t \bullet \exists\, L' \in u \bullet L \downarrow n = L' \downarrow n$. Hence if $\langle t_i \mid i < \omega \rangle$ is a Cauchy sequence in T then there is a corresponding sequence $\langle L_i \mid i < \omega \rangle$ in $\mathbf{S}$ which is Cauchy under the corresponding metric and where $t_i = [L_i]$. There is then a sequence $\langle L'_i \mid i < \omega \rangle$ such that $L_i \sim L'_i$ and $L = \{u \mid \exists\, k \bullet \forall\, i \geqslant k \bullet u \in L'_i\} \in \mathbf{S}$. Then $t = [L]$ is the required limit for $\langle t_i \rangle$. Hence (T, d_T) is a complete metric space.

A.3 Fixed points

Definitions

For any non-empty set J, the space $(J \to T, d_J)$ is a complete metric-space under

$$d_J(a,b) \quad = \quad max\{d_T(a(j), b(j)) \mid j \in J\}$$

A mapping $G : (M_1, d_1) \to (M_2, d_2)$ is *contractive* iff

$$\exists\, c \in [0,1) \bullet \forall\, x, y \in M_1 \bullet d_2(G(x), G(y)) \leqslant c.d_1(x,y)$$

A mapping $G : (M_1, d_1) \to (M_2, d_2)$ is *nonexpansive* iff

$$\forall\, x, y \in M_1 \bullet d_2(G(x), G(y)) \leqslant d_1(x,y)$$

For a in metric space $(J \to T, d_J)$ and $n < \omega$

$$(a \downarrow n)(j) \quad = \quad a(j) \downarrow n$$

defines an operation $(\downarrow n) : (J \to T) \to (J \to T)$.

Mapping $G : (J_1 \to T) \to (J_2 \to T)$ is *nondestructive* iff for $a \in J_1 \to T$ and $n < \omega$

$$G(a) \downarrow n \quad = \quad G(a \downarrow n) \downarrow n$$

Mapping $G : (J_1 \to T) \to (J_2 \to T)$ is *constructive* iff for $a \in J_1 \to T$ and $n < \omega$

$$G(a) \downarrow n+1 \quad = \quad G(a \downarrow n) \downarrow n+1$$

Theorems

The Banach fixed point theorem states that any contractive $G : (M, d) \to (M, d)$ has a unique fixed point in M and $\langle G^i(x) \mid i < \omega \rangle$ converges to that point for any $x \in M$. Moreover, if C is a closed, non-empty subset of M, and C is closed under G, then the fixed point lies in C.

Theorem 9 *For* $G : (J_1 \to T) \to (J_2 \to T)$

$$\begin{array}{rcl} G \text{ is nonexpansive} & \Leftrightarrow & G \text{ is nondestructive} \\ G \text{ is contractive} & \Leftrightarrow & G \text{ is constructive} \end{array}$$

Proof

Omitted. □

Theorem 10 *The operator* $\sqcap : T \times T \to T$ *defined by*

$$t \sqcap u \;=\; \tau.t + \tau.u$$

is constructive and hence contractive.

Proof

Follows directly from the definition and Theorem 9. □

Theorem 11 *For* $A \subseteq \Sigma$*, the operator* $(A \to) : (A \to T) \to T$ *defined by*

$$A \to f \;=\; \Sigma_{a \in A} a.f(a)$$

is constructive and hence contractive.

Proof

Follows directly from the definition and Theorem 9. □

Theorem 12 *The operators defined below are well-defined and nonexpansive.*

$$\begin{array}{rcl} s \,;\, t & = & \Sigma_{x_i \neq \surd} x_i.(s_i \,;\, t) + \Sigma_{x_i = \surd} \tau.t \\ s \mathbin{\triangle} t & = & \Sigma x_i.(s_i \mathbin{\triangle} t) + \Sigma_{y_i = \tau} y_i.(s \mathbin{\triangle} t_i) + \Sigma_{y_i \neq \tau} y_i.t_i \\ s \mid\!\mid\!\mid t & = & \left\{ \begin{array}{l} \Sigma_{x_i \neq \surd} x_i.(s_i \mid\!\mid\!\mid t) \\ + \\ \Sigma_{y_i \neq \surd} y_i.(s \mid\!\mid\!\mid t_i) \\ + \\ \Sigma_{x_i = y_j = \surd} \surd.(s_i \mid\!\mid\!\mid t_j) \end{array} \right. \end{array}$$

$$
\begin{aligned}
s \underset{A}{\parallel} t &= \left\{ \begin{array}{l} \Sigma_{x_i \notin A_\surd} x_i.(s_i \underset{A}{\parallel} t) \\ + \\ \Sigma_{y_i \notin A_\surd} y_i.(s \underset{A}{\parallel} t_i) \\ + \\ \Sigma_{x_i = y_j \in A_\surd} x_i.(s_i \underset{A}{\parallel} t_j) \end{array} \right. \\
s \ {}_A\|_B \ t &= \left\{ \begin{array}{l} \Sigma_{x_i \in A \setminus B} x_i.(s_i \ {}_A\|_B \ t) \\ + \\ \Sigma_{y_i \in B \setminus A} y_i.(s \ {}_A\|_B \ t_i) \\ + \\ \Sigma_{x_i = y_j \in (A \cap B)_\surd} x_i.(s_i \underset{A}{\parallel} t_j) \end{array} \right. \\
s \,\Box\, t &= \Sigma_{x_i \neq \tau} x_i.s_i + \Sigma_{y_i \neq \tau} y_i.t_i + \Sigma_{x_i = \tau} \tau.(s_i \,\Box\, t) + \Sigma_{y_i = \tau} \tau.(s \,\Box\, t_i) \\
s \setminus A &= \Sigma_{x_i \notin A} x_i.(s_i \setminus A) + \Sigma_{x_i \in A} \tau.s_i \\
s [\![R]\!] &= \Sigma\{x.(s_i [\![R]\!]) \mid x \in \Sigma_\tau \wedge ((x_i, x) \in R \vee x_i = x \notin \mathrm{dom}\, R)\} \\
CHAOS(A) &= \tau.STOP + \Sigma_{x \in A_\surd} x.CHAOS(A)
\end{aligned}
$$

Proof

The defining equations determine constructive functions. Hence using Theorem 9 and the Banach fixed point theorem, each defines a unique function. Moreover, since the space of non-expansive operators is a closed subset, and is closed under the defining equations, these functions are also nonexpansive. □

Appendix B

Lexer and parser

B.1 Ambiguity

Common productions

The grammar of Section 7.3 contains multiple paths to the production in *Common* since the general shape of the grammar is

$$\begin{aligned} \mathit{Any} &= \mathit{Proc} \mid \mathit{Expr} \\ \mathit{Expr} &= \mathit{Numb} \mid \mathit{Bool} \mid \mathit{Set} \mid \mathit{Seq} \mid \ldots \\ \mathit{Numb} &= \mathit{Common} \mid \ldots \\ \mathit{Bool} &= \mathit{Common} \mid \ldots \\ \mathit{Set} &= \mathit{Common} \mid \ldots \\ \mathit{Seq} &= \mathit{Common} \mid \ldots \end{aligned}$$

Since every use of the productions which lead directly to *Common* occurs in a guarded context, it is possible to rewrite the grammar to

$$\begin{aligned} \mathit{Any}' &= \mathit{Proc}' \mid \mathit{Expr}' \\ \mathit{Expr}' &= \mathit{Numb}' \mid \mathit{Bool}' \mid \mathit{Set}' \mid \mathit{Seq}' \mid \ldots \\ \mathit{Numb}' &= \ldots \\ \mathit{Bool}' &= \ldots \\ \mathit{Set}' &= \ldots \\ \mathit{Seq}' &= \ldots \end{aligned}$$

provided, for example, every rule involving *Expr* is replaced by corresponding rules involving *Common* and Expr'. The ambiguity then appears as duplicate rules which can be removed from the grammar.

Sequences and comparisons

The grammar of Section 7.3 uses '`<`' and '`>`', both to delimit sequences and as comparison operations. These can usually be distinguished by context, but the use of '`>`' inside sequence comprehensions (as in $\langle n \mid n \leftarrow \langle 1..10 \rangle, n > 5 \rangle$) is ambiguous. This is resolved by tracking additional context information with the lexer and tokenising '`>`' as end-of-sequence if a sequence is potentially active, and as the comparison otherwise.

B.2 Definitions and comments

Definitions in scripts are separated by newlines. However, CSP process descriptions are generally large, often spreading over several lines. It is desirable to provide a mechanism for splitting expressions over lines which does not interfere with the use of newlines to mark the end of each definition. Haskell addresses this issue using indent-rules which are subtle, if effective. A different approach is taken here, using the lexer to 'absorb' newline characters around binary operators; this allows lines to be broken before or after binary operators and has worked well in practice.

The actual implementation of the grammar supports both end-of-line comments (preceded by '`--`') and block comments (enclosed in '`{-`' and '`-}`'.) These are stripped from the input stream by the lexer which treats them as whitespace. The interaction between this and the newline separation of definitions requires significant attention to detail if subtle bugs are to be avoided.

B.3 Flex

This section presents a slightly simplified version of a Flex source file defining the lexer for a version of CSP. Because the file has been simplified, and because the version here will cease to be current, it is recommended that those who want a copy of the file contact the author, rather than using this as a starting point.

The file starts with the definitions of some useful regular expressions. The only ones requiring explanation are `{w}` and `{nl}` which both match arbitrary whitespace, including end-of-line comments. The only difference is that the latter insists on the presence of at east one newline character.

```
alpha           [a-zA-Z]
alnum           [0-9a-zA-Z_]
notalnum        [^0-9a-zA-Z_]
```

```
comment         "--".*
w               ({s}|({comment}[\n]))*
nl              {t}*{comment}?[\n]{w}
s               [\ \r\t\n]
t               [\ \r\t]
int             [0-9]+
primes          [']*
uppers          [A-Z]*
```

Next come the declarations of three exclusive start conditions. These are used to absorb block comments and whitespace.

```
%x              soak
%x              block
%x              sblock
```

The keywords associated with definitions are handled simply using

```
"assert"                { return ASSERT; }
"channel"               { return CHANNEL; }
"datatype"              { return DATATYPE; }
"pragma"{t}.*           { return PRAGMA; }
"transparent"           { return TRANSPARENT; }
"external"              { return EXTERNAL; }
"print"                 { return PRINT; }
```

Names, numeric literals and other simple keywords are equally easy

```
{alpha}{alnum}*{primes} { return NAME; }
{int}                   { return NUMBER; }
"STOP"                  { return STOP; }
"SKIP"                  { return SKIP; }
"true"                  { return KTRUE; }
"false"                 { return KFALSE; }
```

Keywords which correspond to operators are more complex. They are preceded by `{w}` to absorb any whitespace to the left, and switch to the `soak` state to absorb any whitespace on the right.

```
{w}"and"/{notalnum}     { BEGIN soak; return AND; }
{w}"or"/{notalnum}      { BEGIN soak; return OR; }
```

```
{w}"not"/{notalnum}     { BEGIN soak; return NOT; }
{w}"if"/{notalnum}      { BEGIN soak; return IF; }
{w}"then"/{notalnum}    { BEGIN soak; return THEN; }
{w}"else"/{notalnum}    { BEGIN soak; return ELSE; }
{w}"let"/{notalnum}     { BEGIN soak; open_sequence();  return LET; }
{w}"within"/{notalnum}  { BEGIN soak; close_sequence(); return IN; }
```

Like the last two keywords above, the following symbols occur in pairs and interace with the special-casing of '>' within sequence comprehensions. The calls to `open_sequence` indicate to the lexer that a context has been opened within the current one, and a sequence is not currently active. Sequences are marked as active when `LT` is processed in an appropriate context by the parser.

```
"("                     { BEGIN soak; open_sequence();  return OPEN; }
{w}")"                  { close_sequence(); return CLOSE; }
"<"                     { BEGIN soak; return LT; }
{w}">"                  { return active_sequence() ? ENDSEQ : GT; }
"{"                     { BEGIN soak; open_sequence();  return LBRACE; }
{w}"}"                  { close_sequence(); return RBRACE; }
{w}"[["                 { BEGIN soak; open_sequence();  return LSUBST; }
{w}"]]"                 { close_sequence(); return RSUBST; }
"{|"                    { BEGIN soak; open_sequence();  return LPBRACE; }
{w}"|}"                 { close_sequence(); return RPBRACE; }
{w}"["                  { BEGIN soak; return LSQUARE; }
{w}"]"                  { BEGIN soak; return RSQUARE; }
```

The binary operators all follow the same pattern

```
{w}"<="                 { BEGIN soak; return LE; }
{w}">="                 { BEGIN soak; return GE; }
{w}"["{uppers}"="       { BEGIN soak; return REFINED; }
{w}"[|"                 { BEGIN soak; return LCOMM; }
{w}"|]"                 { BEGIN soak; return RCOMM; }
{w}"||"                 { BEGIN soak; return PAR; }
{w}"[>"                 { BEGIN soak; return TIMEOUT; }
{w}"[]"                 { BEGIN soak; return BOX; }
{w}"|~|"                { BEGIN soak; return NDET; }
{w}"\\"                 { BEGIN soak; return BACKSLASH; }
{w}"|"                  { BEGIN soak; return PIPE; }
{w}"|||"                { BEGIN soak; return INTL; }
{w}"->"                 { BEGIN soak; return ARROW; }
```

```
{w}"<-"                 { BEGIN soak; return BECOMES; }
{w}":"                  { BEGIN soak; return COLON; }
{w}"/\\"                { BEGIN soak; return INTR; }
{w}"&"                  { BEGIN soak; return GUARD; }
{w}";"                  { BEGIN soak; return SEMI; }
{w}"."                  { BEGIN soak; return DOT; }
{w}"?"                  { BEGIN soak; return QUERY; }
{w}"!"                  { BEGIN soak; return PLING; }
{w}"@"                  { BEGIN soak; return AT; }
{w}","                  { BEGIN soak; return COMMA; }
{w}"="                  { BEGIN soak; return EQUAL; }
{w}"+"                  { BEGIN soak; return PLUS; }
"-"                     { BEGIN soak; return MINUS; }
{w}"*"                  { BEGIN soak; return TIMES; }
{w}"/"                  { BEGIN soak; return SLASH; }
{w}"%"                  { BEGIN soak; return MOD; }
{w}"#"                  { BEGIN soak; return HASH; }
{w}"=="                 { BEGIN soak; return EQ; }
{w}"!="                 { BEGIN soak; return NE; }
{w}"^"                  { BEGIN soak; return CAT; }
{w}".."                 { BEGIN soak; return DOTDOT; }
```

Whitespace which is not absorbed by the operators is matched by one of the first two rules here. The latter catches any characters in the script which were unexpected.

```
{t}                     { }
{nl}                    { BEGIN soak; return NEWLINE; }
.                       { BADCHAR }
```

The `soak` state consumes whitespace after binary operators. A similar effect can be achieved using a trailing instance of `{w}`, but produces much larger lexers.

```
<soak>{w}               { }
<soak>"{-"              { ++block_comment_count; BEGIN sblock; }
<soak>""/.              { BEGIN 0; }
```

The `block` state discards block comments, handling nested comments. The `sblock` state performs the same task for the `soak` state. The two differ only in the state which is selected on exit.

```
"{-"                    { ++block_comment_count; BEGIN block; }
```

```
<block>"{-"             { ++block_comment_count; }
<block>.                { }
<block>[\n]             { }
<block>{comment}        { /* Ignore end comments in EOL comments */ }
<block>"-}"             { if(--block_comment_count == 0) BEGIN 0; }

<sblock>"{-"            { ++block_comment_count; }
<sblock>.               { }
<sblock>[\n]            { }
<sblock>{comment}       { /* Ignore end comments in EOL comments */ }
<sblock>"-}"            { if(--block_comment_count == 0) BEGIN soak; }
```

B.4 Bison

This section presents a slightly simplified version of a Bison source file defining the parser for a version of CSP. Again, it is recommended that those who want a version of the parser contact the author. The type definitions and calls needed to construct the syntax-tree have been omitted for brevity.

The file starts by defining the tokens to be returned from the lexer. The ordering of the tokens indicates their precedence. The definitions also indicate the associativity of the tokens.

```
%token CHAOS STOP SKIP LBRACE RBRACE LPBRACE RPBRACE EQUAL BECOMES
%token COMMA DOTDOT PIPE NAME NUMBER NEWLINE KTRUE KFALSE PRAGMA
%token ASSERT CHANNEL DATATYPE REFINED TRANSPARENT EXTERNAL PRINT

%nonassoc LET IN LAMBDA LAMBDADOT IF THEN ELSE
%nonassoc COLON AT
%left BACKSLASH
%left INTL
%nonassoc LCOMM RCOMM LSQUARE PAR RSQUARE
%left NDET
%left BOX
%left INTR
%left TIMEOUT
%left SEMI
%nonassoc GUARD
%right ARROW QUERY PLING
%right DOT
```

```
%left OR
%left AND
%left NOT
%nonassoc EQ NE LT LE GT GE ENDSEQ
%left PLUS MINUS
%left TIMES SLASH MOD
%left HASH
%left CAT
%token OPEN CLOSE LSUBST RSUBST
```

A script, as supplied to the parser, is to be a list of definitions, separated by newlines. Some definitions are also permitted inside let/within blocks.

```
%start begin

%%

script          : newlines0 defns newlines0
                | newlines0
                ;

simple_defns    : simple_defn
                | simple_defns newlines simple_defn
                ;

defns           : defn
                | defns newlines defn
                ;

simple_defn     : defn_left EQUAL any
                | TRANSPARENT names
                | EXTERNAL names
                ;

defn            : ASSERT check
                | CHANNEL names COLON seqtype
                | CHANNEL names
                | DATATYPE name EQUAL vartype
                | PRINT any
                | pragma
                | simple_defn
```

```
                ;

check           : bool
                | proc refined proc
                ;
```

As presented here these rules make the grammar ambiguous. In practice these productions are eliminated from the grammar by a pre-processor which expands them wherever they occur.

```
any             : _proc
                | expr
                ;
expr            : _dotted
                | _bool
                | _numb
                | _set
                | _seq
                | _tuple
                | amb
                ;
bool            : _bool | amb ;
dotted          : _dotted | amb ;
numb            : _numb | amb ;
proc            : _proc | amb ;
seq             : _seq  | amb ;
set             : _set  | amb ;
```

We restrict the expressions which can occur on the left of a definition

```
defn_call_left  : name
                | defn_call_left OPEN CLOSE
                | defn_call_left OPEN anys CLOSE
                ;

defn_left       : defn_call_left
                | _tuple
                | _seq
                | _set
                | _dotted
                | _bool
                ;
```

The productions for boolean, dotted, number, set, sequence and process terms are as given in Chapter 7.

```
_bool           : NOT bool
                | bool AND bool
                | bool OR bool
                | expr EQ linkexpr
                | expr NE linkexpr
                | expr LE linkexpr
                | expr LT linkexpr
                | expr GE linkexpr
                | expr GT linkexpr
                | KTRUE
                | KFALSE
                ;

_dotted         : expr DOT expr
                ;

_tuple          : OPEN any COMMA anys CLOSE
                ;

_numb           : literal
                | numb PLUS numb
                | numb MINUS numb
                | MINUS numb
                | numb TIMES numb
                | numb MOD numb
                | numb SLASH numb
                | HASH seq
                ;

_seq            : begseq targ0 optgens endseq
                | seq CAT seq
                ;

_set            : LBRACE targ0 optgens RBRACE
                | LPBRACE exprs0 optgens RPBRACE
                ;
_proc           : PAR gens AT LSQUARE set RSQUARE proc %prec AT
                | NDET gens AT proc                    %prec AT
```

```
                | BOX gens AT proc                        %prec AT
                | INTL gens AT proc                      %prec AT
                | SEMI gens AT proc                      %prec AT
                | LCOMM set RCOMM gens AT proc          %prec AT
                | proc BACKSLASH set
                | proc INTL proc
                | proc LCOMM set RCOMM proc
                | proc LSQUARE set PAR set RSQUARE proc
                | proc NDET proc
                | proc BOX proc
                | proc TIMEOUT proc
                | proc INTR proc
                | proc SEMI proc
                | bool GUARD proc
                | dotted fields ARROW proc
                | dotted ARROW proc
                | STOP
                | SKIP
                | CHAOS OPEN set CLOSE
                ;
```

The ambiguous terms in the grammar include those from the *Lambda* production of Chapter 7, as well as the renaming operator. The former is a simplification, while the latter is necessary to give renaming a high precedence.

```
amb             : IF bool THEN any ELSE any
                | LET simple_defns newlines0 IN any
                | BACKSLASH args AT any
                | amb AS any
                | simple
                ;

simple          : name
                | simple OPEN args CLOSE
                | simple LSUBST pairs0 optgens RSUBST
                | OPEN any CLOSE
                ;
```

Type expressions are given by

```
type            : _set
```

```
                | OPEN type COMMA types CLOSE
                | name
                | name OPEN types CLOSE
                ;

seqtype         : type
                | type DOT seqtype
                ;

vartype         : branch
                | vartype PIPE branch
                ;

branch          : name
                | name DOT seqtype
                ;
```

Fields and comprehension targets are precisely as given in Chapter 7.

```
field           : QUERY expr
                | QUERY expr COLON set
                | PLING expr
                ;

targ            : anys
                | numb DOTDOT
                | numb DOTDOT numb
                ;
```

The remaining productions are trivial

```
args            :
                | anys
                ;

gen             : expr BECOMES expr
                | bool
                ;

pair            : dotted BECOMES dotted
                ;
```

```
targ0           :
                | targ
                ;


/* Simple tokens, wrapped to syntax */

name            : NAME
                ;

literal         : NUMBER
                ;

pragma          : PRAGMA
                ;

refined         : REFINED
                ;

/* Patch for ambiguity of > */

begseq          : LT
                ;

endseq          : ENDSEQ
                ;


/* Trivial lists of items */

anys            : any
                | any COMMA anys
                ;

exprs           : expr
                | expr COMMA exprs
                ;

exprs0          :
```

```
                | exprs
                ;

types           : type
                | type COMMA types
                ;

fields          : field
                | field fields
                ;

gens            : gen
                | gen COMMA gens
                ;

optgens         :
                | PIPE gens
                ;

names           : name
                | name COMMA names
                ;

newlines        : NEWLINE newlines0
                ;

newlines0       :
                | newlines
                ;

pairs           : pair
                | pair COMMA pairs
                ;

pairs0          :
                | pairs
                ;
```

Bibliography

[1] R. Allen. *A Formal Approach to Software Architecture.* Carnegie Mellon, School of Computer Science PhD Thesis, 1997. Issued as CMU Technical Report CMU-CS-97-144.

[2] R. Allen and D. Garlan. *A Formal Basis for Architectural Connection.* ACM Transactions on Software Engineering and Methodology, July 1997.

[3] G. Barrett, D. May and D. Shepard. *Formal methods in the design of the T9000.* Transputer Systems–Ongoing Research (Proceedings of the 15th WoTuG Technical Meeting). IOS Press, 1992.

[4] G. Barrett. *Model Checking in Practice: The T9000 Virtual Channel Processor.* Formal Methods Europe 1993: Industrial-Strength Formal Methods. Springer Verlag, LNCS 670.

[5] G. Barrett. occam3 *language definition.* Inmos draft paper.

[6] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[7] S.D. Brookes, A.W. Roscoe, and D.J. Walker. *An Operational Semantics for CSP.* Unpublished report.

[8] Bettina Buth, Michael Kouvaras, Jan Peleska, and Hui Shi. *Deadlock Analysis for a Fault-Tolerant System.* Proceedings of AMAST'97, Springer LNCS 1349, 1997.

[9] Bettina Buth, Jan Peleska, and Hui Shi. *Combining Methods for the Livelock Analysis of a Fault-Tolerant System.* Submitted to AMAST'98, 1998.

[10] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems.* ACM Transactions of Programming Languages and Systems, 15(1):36-72, January 1993.

[11] R. Cleaveland and S. Sims. *The Concurrency Workbench of North Carolina (Version 1.1) User's Manual.* Available from `http://www.csc.ncsu.edu/eos/users/r/rance/WWW/cwb-nc.html`.

[12] S. Creese. *An inductive technique for modelling arbitrarily configured networks.* Oxford University MSc Thesis, 1997.

[13] J.W.M. Davies. *Specification and Proof in Real-Time Systems.* Cambridge University Press, 1993.

[14] Formal Systems (Europe) Limited. *Failures-Divergence Refinement: User Manual and Tutorial.* Available by FTP from Oxford University.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[16] M.H. Goldsmith, A.W. Roscoe and B.G.O. Scott. *Denotational semantics for* occam 2. PRG Technical Monograph 108.

[17] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual.* Prentice-Hall, 1984, 1987, 1991, 1995.

[18] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[19] Inmos Limited. occam 2 *Reference Manual.* Prentice-Hall, 1988.

[20] D.M. Jackson. *The Specification of Aircraft Engine Control Software in Timed CSP.* Oxford University M.Sc. thesis, 1989.

[21] Mark Jones. *An Introduction to Gofer.* Part of the Gofer 2.30 distribution.

[22] R.S. Lazić. *A semantic study of data-independence with applications to model-checking.* Oxford University D.Phil Thesis, 1998.

[23] G. Lowe. *Breaking and fixing the Needham-Schroeder public-key protocol using FDR.* Proceedings of TACAS'97, Springer LNCS 1055, 1996.

[24] G. Lowe. *Casper: a compiler for the analysis of security protocols.* Proceedings of 1997 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1997.

[25] G. Lowe. *Pravda manual.* Available by FTP from Oxford University.

[26] J.M.R. Martin. *The design and construction of deadlock-free concurrent systems.* University of Buckingham D.Phil thesis, 1996.

[27] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[28] Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* Aarhus University Report, 1981.

[29] J.N. Reed, D.M. Jackson, B. Deianov, and G.M. Reed. *Automated Formal Analysis of Networks: FDR Models of Arbitrary Topologies amd Flow-Control Mechanisms.* European Joint Workshop on Theory and Practice in Software; Fundamental Approaches to Software Engineering (ETAPS-FASE'98).

[30] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1998.

[31] A.W. Roscoe. *Two Papers on CSP.* PRG Monograph 67. Oxford University, 1988.

[32] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. *Non-interference through determinism.* Journal of Computer Security 4.1, 1996. Revised from Proceedings of the European Symposium on Research in Computer Security (ESORICS) 1994, LNCS 875.

[33] J.B. Scattergood. *Parsing and Type-Checking CSP.* Oxford University Transfer Thesis, 1992.

[34] J.B. Scattergood and K. Seidel. *Translating* occam 2 *to CSP.* Transputer Applications and Systems, 1994. IOS Press.

[35] A. Simpson. *Safety through Security.* Oxford University D.Phil Thesis, 1996.

[36] P. Steggles and J.R. Hulance. **Z** *Tools Survey.* Technical Report, Imperial Software Technology/FSEL, 1994.

[37] Bjarne Stroustrup. *The C++ Programming Language (Second Edition).* Addison-Wesley, 1992.

[38] L. Wulf. *Interaction and Security in Distributed Computing.* Oxford University D.Phil Thesis, 1997.

[39] J.T. Yantchev. *ARC–A tool for efficient refinement and equivalence checking for CSP.* IEEE 2nd International Conference on Algorithms and Architectures for Parallel Processing, 1996.